Pune Vidyarthi Griha's

COLLEGE OF ENGINEERING, NASHIK – 3.

"Linear Data Structure using Squential Organization"

By

Prof. Anand N. Gharu

(Assistant Professor)

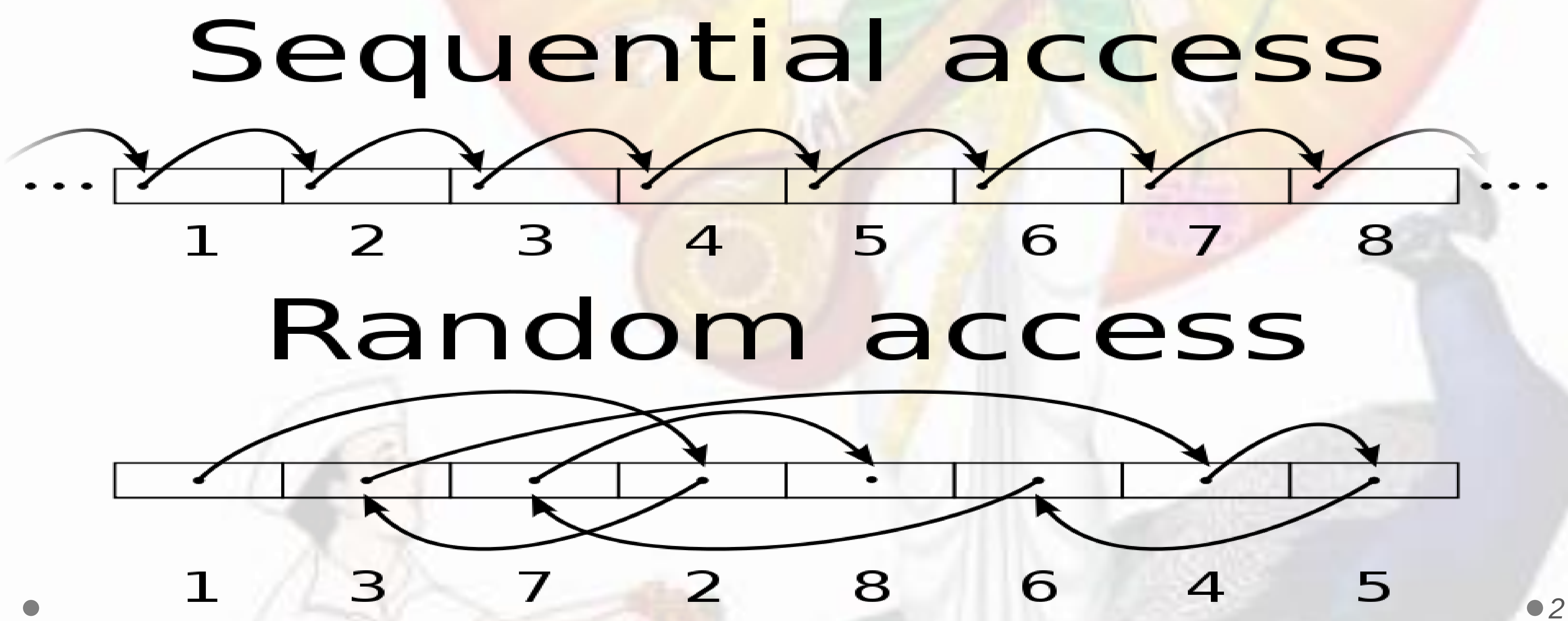
PVGCOE Computer Dept.

10 July 2019

SEQUENTIAL ORGANIZATION

- **Definition :**

“In computer science, **sequential** access means that a group of elements (such as **data** in a memory array or a disk file or on magnetic tape **data** storage) is accessed in a predetermined, ordered sequence.**Sequential** access is sometimes the only way of accessing the **data**, for example if it is on a tape.”



Linear Data Structure Using Sequential Organization

- **Definition :**

“The data structure where data items are organized sequentially or linearly one after another is called as **Linear Data Structure**”

A **linear data structure**traverses the **data** elements sequentially, in which only one **data** element can directly be reached. Ex: Arrays, Linked Lists.

Array

- **Definition :**

❖ "An array is a finite ordered **collection of homogeneous data elements** which provides **direct access** (or random access) to any of its elements.

❖ An array as a **data structure** is defined as a set of pairs (index,value) such that with each index a value is associated.

- **index** — indicates the location of an element in an array.
- **value** - indicates the actual value of that data element.

❖ Declaration of an array in 'C++':

- **int Array_A[20];**

"

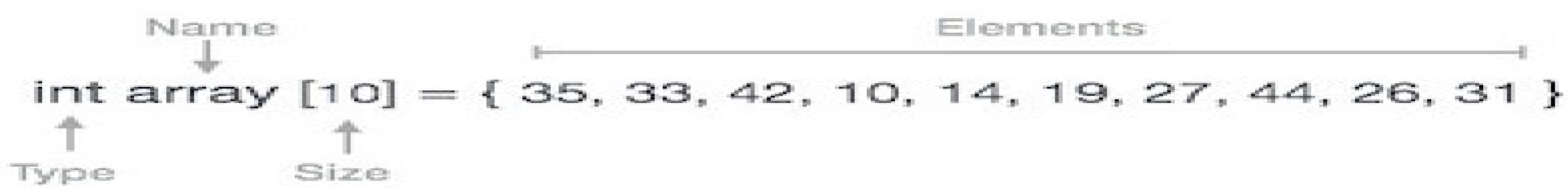
Array

- **Array Representation**
- Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

- Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.
- As per the above illustration, following are the important points to be considered.

Array

- **Array Representation**
- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.
- Basic Operations
- Following are the basic operations supported by an array.
- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

Memory Representation and Calculation

A. Memory Representation

– Memory arrangement after declaring one dimensional array shown in Fig. 2.3.1.

		Memory Address
percentage [0]		3000
percentage [1]		3004
percentage [2]		3008
percentage [3]		3012
percentage [4]		3016
percentage [5]		3020
percentage [6]		3024
percentage [7]		3028
percentage [8]		3032
percentage [9]		3036

Fig. 2.3.1

– Initialize an array to store percentage of 10 students

```
percentage[0] = 99.07;
percentage[1] = 79.47;
percentage[2] = 60.60;
percentage[3] = 54.04;
percentage[4] = 80.60;
percentage[5] = 91.70;
percentage[6] = 90.30;
percentage[7] = 85.27;
percentage[8] = 99.70;
percentage[9] = 94.50;
```

ADDRESS CALCULATION

- The address of the i^{th} element is calculated by the following formula

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

  Here, base address is the address of the first element where array storage starts.

Address calculation of 1D array

Address of arr[i] = base address + i * element_size

Address of percentages[3] = 3000 + 3 * sizeif(float)

= 3000 + 3 * 4 = 3012

Abstract Data Type

- ADT is useful tool for specifying the logical properties of a data type.
- A data type is a collection of values & the set of operations on the values.
- ADT refers to the mathematical concept that defines the data type.
- ADT is not concerned with implementation but is useful in making use of data type.

ADT for an array

- Arrays are stored in consecutive set of memory locations.
- Array can be thought of as set of index and values.
- For each index which is defined there is a value associated with that index.
- There are two operations permitted on array data structure .retrieve and store

ADT for an array

- CREATE()-produces empty array.
- RETRIVE(array,index)->value

 Takes as input array and index and either returns

 appropriate value or an error.

- STORE(array,index,value)-array used to enter new index

 value pairs.

Introduction to arrays

Representation and analysis

Type variable_name[size]

Operations with arrays:

Copy

Delete

Insert

Search

Sort

Merging of sorting arrays.

Copy operation

```
#include <stdio.h>

int main()
{
   int a[100],b[100] position, c n;

   printf("Enter number of elements in array\n");
   scanf("%d", &n);

   printf("Enter %d elements\n", n);

   for ( c = 0 ; c < n ; c++ )
      scanf("%d", &a[c]);

  printf("Enter %d elements\n", n);


      for( c = 0 ; c < n - 1 ; c++ )
         printf("%d\n", a[c]);

//Coping the element of array a to b

  for( c = 0 ; c < n - 1 ; c++ )
     {
         b[c]=a[c];
     }
   }

   return 0;
}
```

Output

- Enter number of elements in array -4
- Enter 4 elements

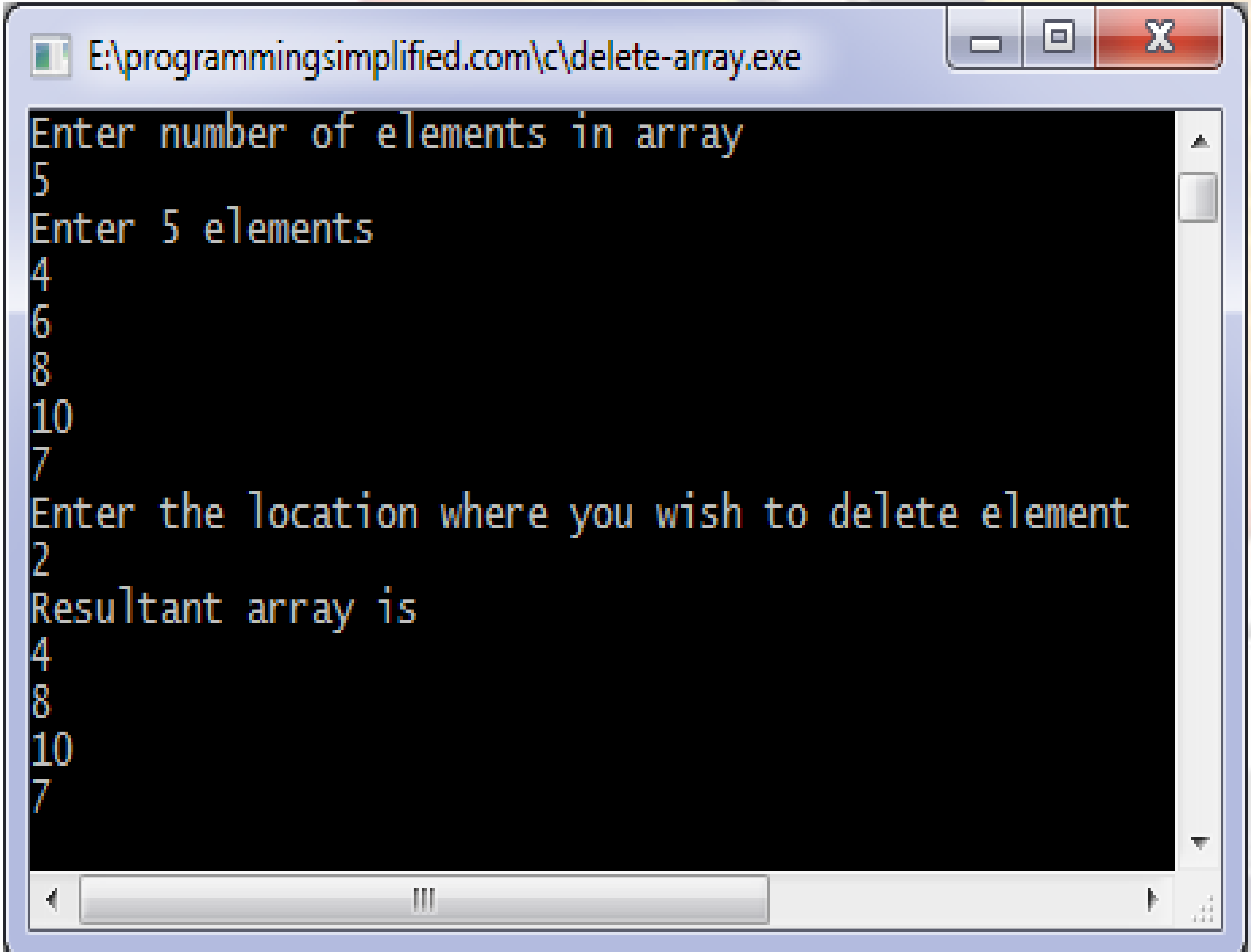
1
- 2
- 3
- 4

- displaying array a
- 1
- 2
- 3
- 4
- displaying array b
- 1
- 2
- 3
- 4

Delete operation

```
#include <stdio.h>

int main()
{
  int array[100], position, i, n;

  printf("Enter number of elements in array\n");
  scanf("%d", &n);

  printf("Enter %d elements\n", n);

  for ( i = 0 ; i < n ; i++ )
    scanf("%d", &array[i]);

  printf("Enter the location where you wish to delete element\n");
  scanf("%d", &position);

    for ( i = position ; i < n; i++ )
      {

      array[i] = array[i+1];

     }
     printf("Resultant array is\n");

      for( i = 0 ; i < n-1  ; i++ )

      printf("%d\n", array[i]);
  return 0;
}
```

Delete operation



Inserting an element

```
#include <stdio.h>

int main()
{
  int array[100], position, i, n, value;

  printf("Enter number of elements in array\n");
  scanf("%d", &n);

  printf("Enter %d elements\n", n);

  for (i= 0;i< n; i++)
    scanf("%d", &array[i]);

  printf("Enter the location where you wish to insert an element\n");
  scanf("%d", &position);

  printf("Enter the value to insert\n");
  scanf("%d", &value);

  for (i = n - 1; i >= position ; i--)
    array[i+1] = array[i];

  array[position] = value;

  printf("Resultant array is\n");

  for (i= 0; i <= n; i++)
    printf("%d\n", array[i]);
  return 0;
```

Inserting an element

E:\programmingsimplified.com\c\insert-array.exe

```
Enter number of elements in array
5
Enter 5 elements
2
5
4
3
8
Enter the location where you wish to insert an element
4
Enter the value to insert
10
Resultant array is
2
5
4
10
3
8
```

Sort an array

```
Int a[10]={5,4,3,2,1}
for(i=0;i<n-1;i++)
{
 for(j=0;j<=n-1;j++)
 {
      if(a[j]>a[j+1])
         {
            temp=a[i];
              a[i]=a[j];
              a[j]=temp;
            }
}
```

Reverse array

```
#include <stdio.h>

int main() {
  int array[100], n, i, temp, end;

  scanf("%d", &n);
  end = n - 1;

  for (i = 0; i < n; i++) {
    scanf("%d", &array[i]);
  }

  for (i= 0; < n/2; i++)
 {
     temp  = array[i];
    array[i]   = array[end];
    array[end] = temp;

    end--;
  }

  printf("Reversed array elements are:\n");

  for ( i= 0; i < n; i++) {
    printf("%d\n", array[i]);
  }

  return 0;
}
```

Sort element using array

```
int a[10]={5,4,3,2,1}
for(i=0;i<n;i++)
  for(j=i+1;j<n;j++)
  {
      if(a[i]>a[j])
        {
          temp=a[i];
           a[i]=a[j];
           a[j]=temp;
         }
}
```

Two-dimensional Arrays in C

- **multidimensional array is the two-dimensional array**

- **type arrayName [x][y];**

Two-dimensional Arrays in C

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

```
m-no of rows
n-no of columns
Printf("\n Enter the rows and columns");
Scanf(%d %d",&m,&n);
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
        {
        Printf("\n Enter the value of(%d)(%d)=",i,j);
        Scanf("%d",&a[i][j]);
        }
        }
```

```
for(i=0;i<m;i++)
{
   Printf("\n");
    for(j=0;j<n;j++)
   {
      printf("%d",&a[i][j]);
   }
   }
```

ARRAY AS AN ADT

- Formally ADT is a collection of domain, operations, and axioms (or rules)
- For defining an array as an ADT, we have to define its very basic operations or functions that can be performed on it
- The basic operations of arrays are creation of an array, storing an element, accessing an element, and traversing the array

List of operation on Array :-

- **1.** Inserting an element into an array
- 2. deleting element from array
- 3. searching an element from array
- 4. sorting the array element

N -dimensional Arrays

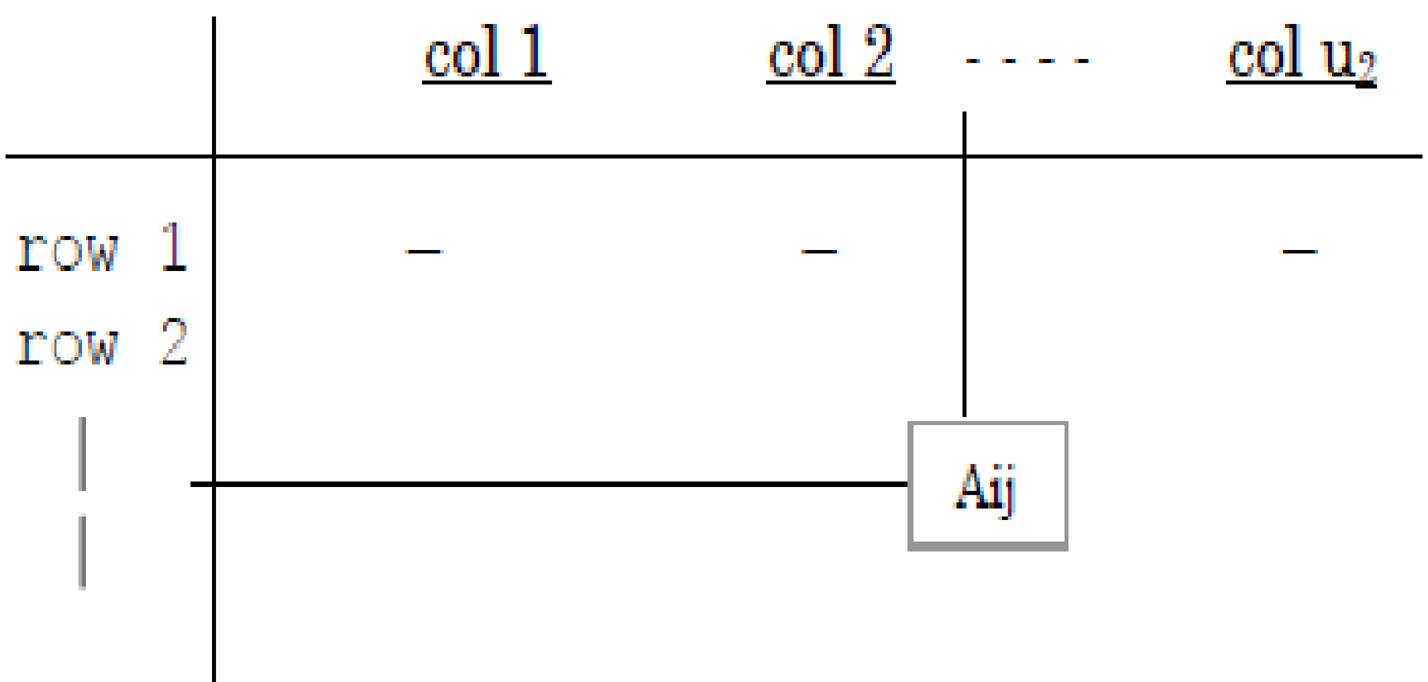
1D (One Dimension):

Let A[m_1] be a one-dimensional array. Let A[0] be stored at address Base= X. Now assuming one element per location, the address of A[1] is X+1, address of an arbitrary element A[i] is given by X + i, and the address of A[m_1-1] is X+ m_1-1.

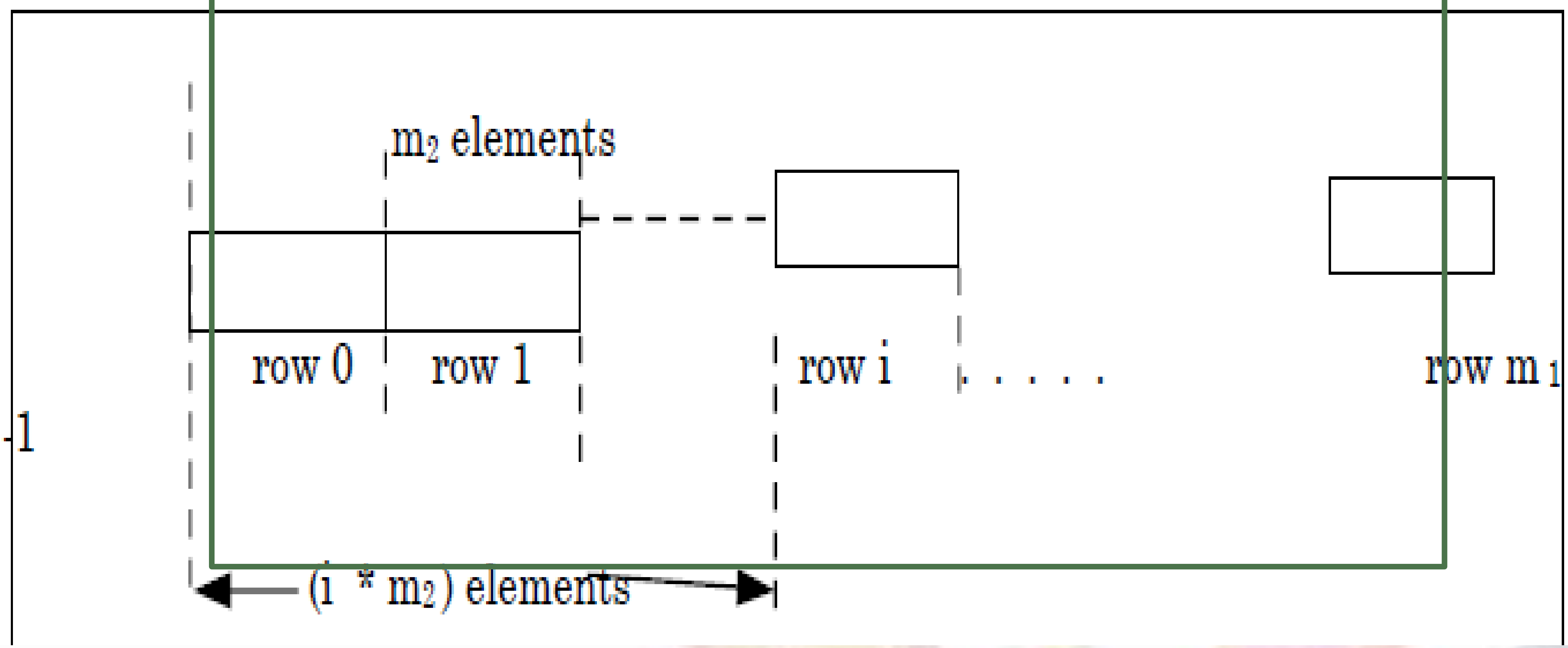
A[0]	A[1]	A[2]	....		A[i]	....	A[m_1-1]
X	X+1	X+2	....		X+i	....	X + (m_1 – 1)

2D(Two Dimension):

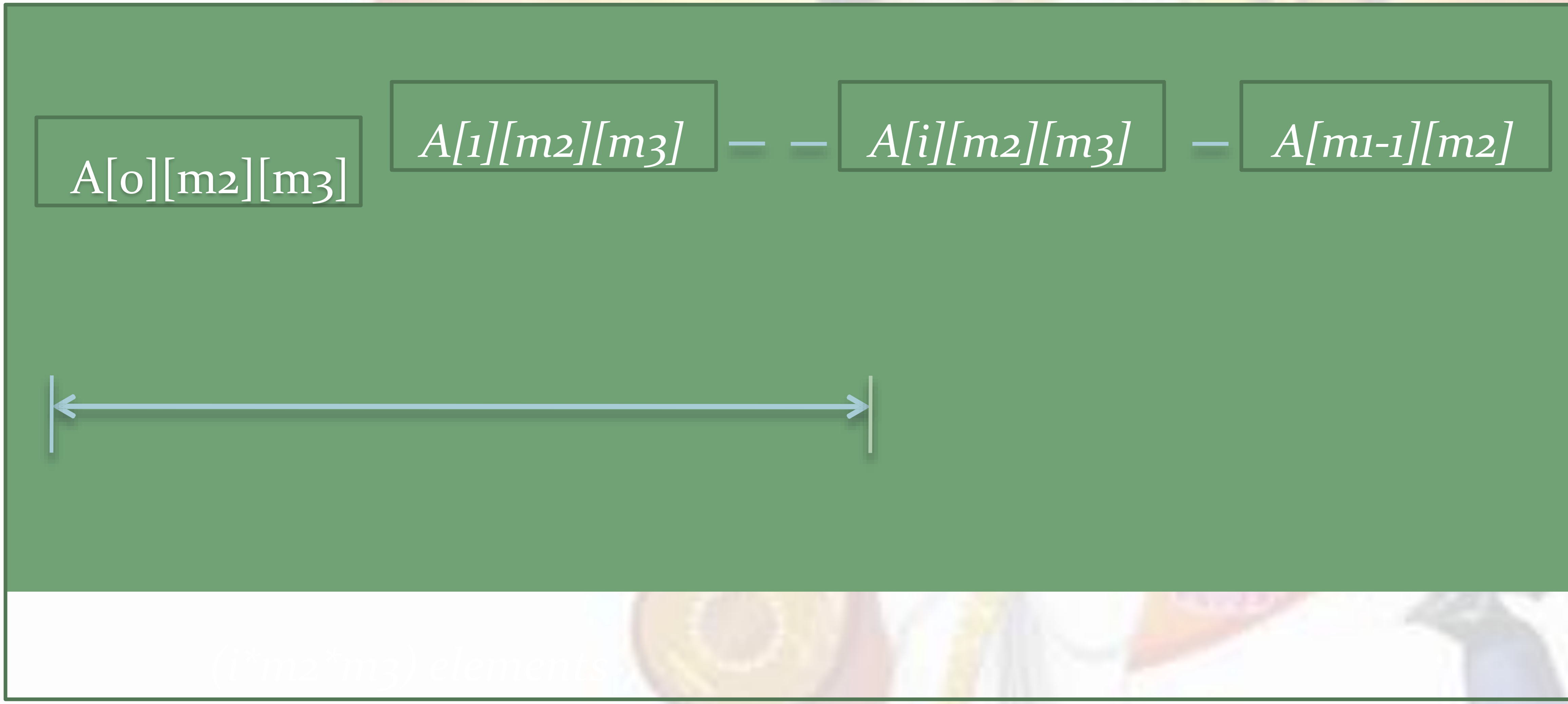
Now consider two-dimensional array $A[m_1][m_2]$ which has m_1 rows as row_1, row_2 - - - - row m_1-1 and each row contains m_2 elements as there are m_2 columns.



Now let A[0][0] be stored at address X then A[0][1] would be stored at X+1; A[0][i] would be at X+i and so on till A[0][m_2-1] at X + (m_2-1). Now address of A[i][0] would be X+ (i* m_2). Address of A[i][j] is X+(i * m_2) +j.



Row-Major representation of 2D array



Three dimensions row-major arrangement

- ***The address of A[i][j][k] is computedas***

  - *Addrof A[i][j][k] = X + i * m2 * m3 + j * m3 + k*
  - *By generalizing this we get the address of **A[i1][i2][i3] … [in]** in n-dimensional array A[m1][m2][m3]. ….[mn]*
  - *Consider the address of **A [0][0][0]…..[0]** is X then the address of **A [i][0][0]….[0] = X + (i1 * m2 * m3 * - - -- - * mn)** and*
  - ***Address of A** [i1][i2] …. [0] = X + (i1 * m2 * m3 * - -- - *mn) + (i2 * m3 * m4 *--- * mn)*
- *Continuing in a similar way, address of **A[i1][i2][i3]**- - - -[**in]** willbe*
- *Address of A[i1][i2][i3]----[in] = X + (i1 * m2 * m3 * - - -- - * mn) +*

*(i2 * m3 * m4 *--- - - * mn)+(i3 * m4 * m5--- * mn + (i4 * m5 * m6--*

*- - - * mn +…….+ in* = $X + \sum_{j=1}^{n} i_j * A_j$

Ex-Arrays

```
#include <stdio.h>
int main ()
{
    int a[10],i,size;

        printf(“\nhow many no of elements u want to  scan”);

         scanf(“%d”,&size);

         printf(“\nEnter the elements in  the array”);

          for(i=0;i<size;i++)
          {
                                scanf(“%d”,&a[i]);

            }    //end for

        for(i=0;i<size;i++)
        {
                      printf(“The array is %d”,a[i]); //Displaying Array

           }    //end for

 return 0;
```

Output will be

1
2
3
4
5

Multi-dimensional Arrays in C

- type name[size1][size2]...[sizeN];

Two-dimensional Arrays in C

- multidimensional array is the two-dimensional array
- type arrayName [x][y];

Two-dimensional Arrays in C

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

HOW TO INITIALIZE 2-D ARRAY IN PROGRAM

- Initializing Two-Dimensional Arrays

int a[3][4] = { {0, 1, 2, 3} , /* initializers for

{4, 5, 6, 7} ,

{8, 9, 10, 11} /* initializers for row

/* initializers for row indexed by 2 */ };

- **Accessing Two-Dimensional Array Elements**

```
int  val = a[2][3];
```

Three-dimensional Arrays in C

- For example, the following declaration creates a three dimensional integer array –

- Ex-int threedim[5][10][4];

THE CLASS ARRAY

- ***Arrays support various operations such as traversal, sorting, searching, insertion, deletion, merging, block movement, etc.***

- *Insertion of an element into an array*
- *Deleting an element*
- *Memory Representation of Two-Dimensional Arrays*

Row-major Representation
Column-major Representation

Columns →

$$\begin{array}{ll} & \begin{array}{cccc} \text{Col1} & \text{col2} & .... & \text{coln} \end{array} \\ \begin{array}{l} \text{Rows } R_1 \\ \\ R \\ \\ \\ \\ \end{array} & \begin{bmatrix} A_{11} & A_{12} & .... & A_{1n} \\ A_{11} & A_{12} & .... & A_{1n} \\ : & : & & : \\ A_{m1} & A_{m2} & .... & A_{mn} \end{bmatrix} \end{array}$$

$m*n$

$$\text{Matrix } M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

Row-major representation

Column Index

Row Index	0	1	2	3
0	8	6	5	4
1	2	1	9	7
2	3	6	4	2

Two-Dimensional Array

– For this array the row-major representation will be :

8	6	5	4	2	1	9	7	3	6	4	2
Row 0				Row 1				Row 2			

– For column-major order, all elements of the first column come before all elements of the second column, etc.

– For the given array the column-major representation will be :

8	2	3	6	1	6	5	9	4	4	7	2
Column 0			Column 1			Column 2			Column 3		

☞ **Address Calculation of 2D array**

A. Row major representation

Address of [i][j] = base address + i *c * element_size
+ j * element_size
= base address + (i*c*+j) * element_size

Example

– Adresss of arr[1][2] (Consider base address as 1000)

Arr[1][2 = 1000 + (1*3+2) * sizeof(int)
= 1000 + 10 = 1010

B. Column major representation

Address of [i][j]= base address + j *r * element_size
+ i * element_size
= base address + (j*r*+i) * element_size

Example

– Adresss of arr[1][2] (Consider base address as 1000)

Arr[1][2] = 1000 + (2*4+1) * sizeof(int)
= 1000 + 18 = 1018

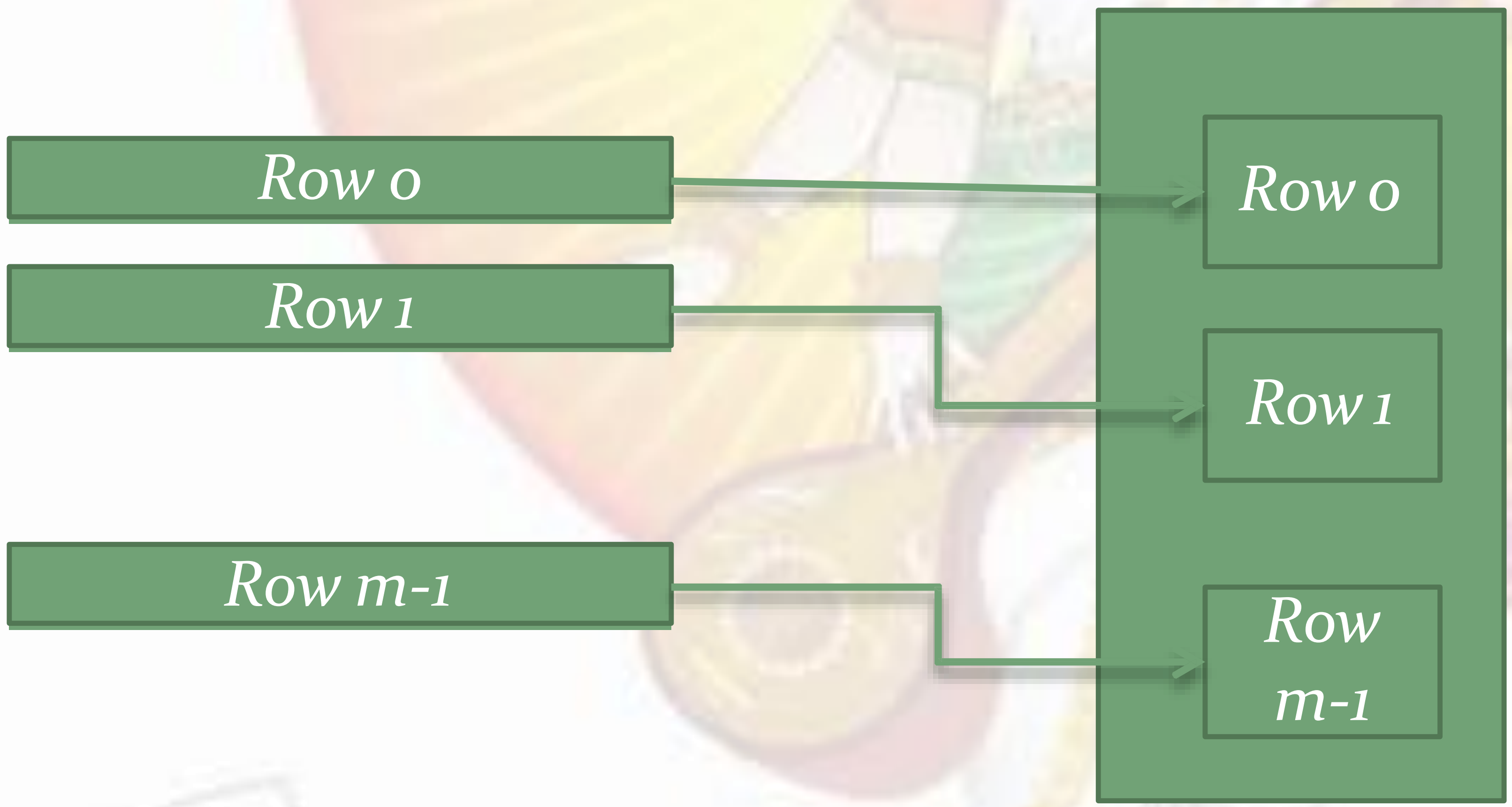
Row-major representation

- *In row-major representation, the elements of Matrix are stored row-wise, i.e., elements of 1st row, 2nd row, 3rd row, and so on till mth row*

1	2	3	4	5	6	7	8	9	10	11	12
(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)	(2,0)	(2,1)	(2,2)	(2,3)

Row1 Row2 Row3

Row major arrangement



Row-major arrangement in memory , in row major representation

- *The address of the element of the ith row and the jth column for matrix of size m x n can be calculated as:*

Addr(A[i][j]) = Base Address+ Offset = Base Address + (number of rows placed before ith row * size of row) * (Size of Element) + (number of elements placed before in jth element in ith row)* size of element

- *As row* ***indexing starts from 0****, i indicate number of rows before the ith row hereand similarly for j.*

For Element Size = 1 the address is

Address of A[i][j]= Base + (i * n) +j

- *In general,*

$$Addr[i][j] = ((i-LB_1) * (UB_2 - LB_2 + 1) * size) + ((j - LB_2) * size)$$

  - *where number of rows placed before ith row = $(i - LB_1)$*
  - *where LB_1 is the lower bound of the first dimension.*

*Size of row = (number of elements in row) * (size of element)Memory Locations*

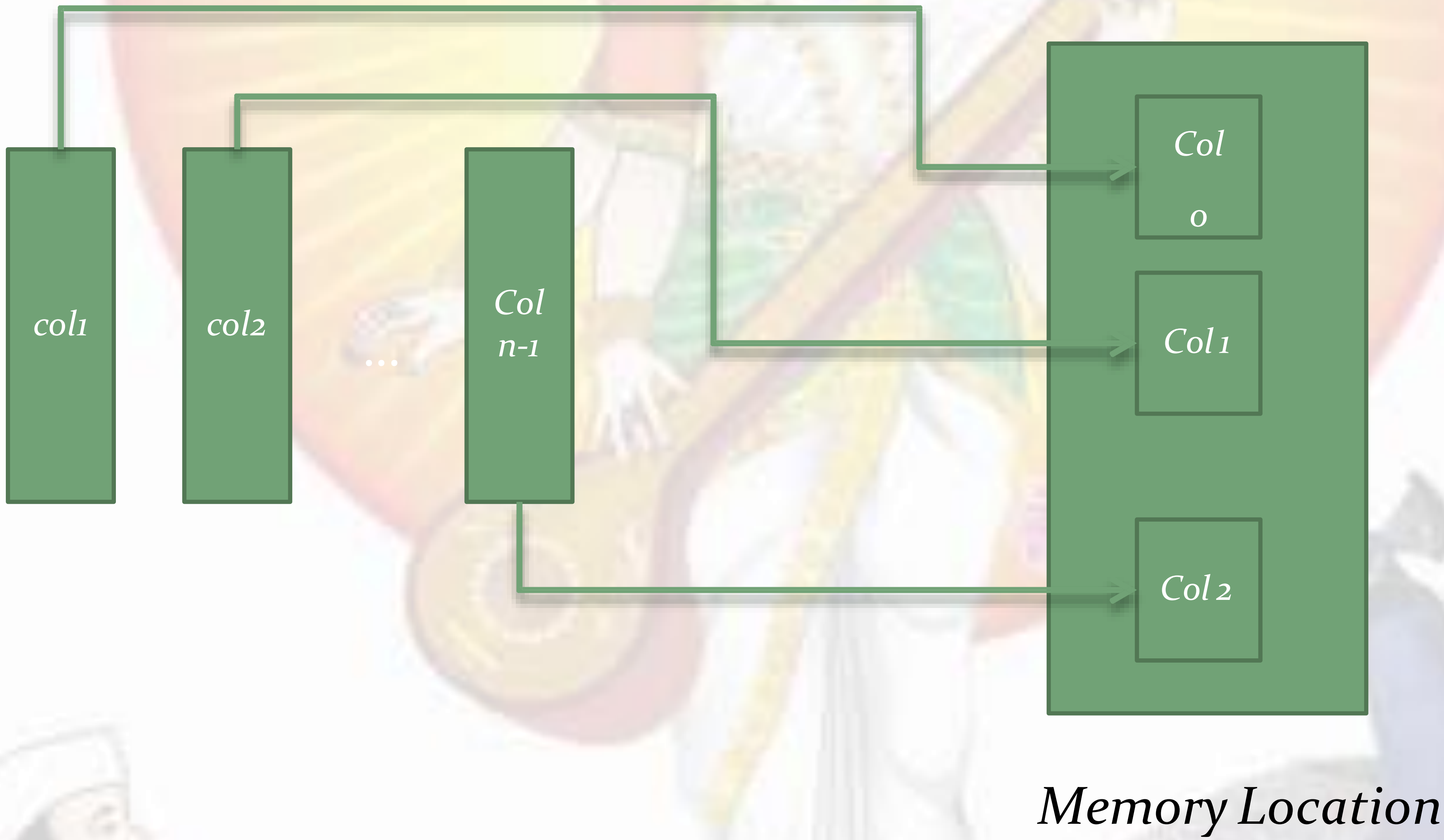
The number of elements in a row = $(UB_2 - LB_2 + 1)$

  - *where UB_2 and LB_2 are upper and lower bounds of the second dimension.*

Column-major representation

- *In **column-major representation** m × n elements of two- dimensional array A are stored as one single row of columns.*
- *The elements are stored in **the memory as a sequence** as first the elements of column 1, then elements of column 2 and so on till elements of column n*

Column-major arrangement



Column-major arrangement in memory , in column major representation

- *The address of A[i][j] is computed as*
  - ***Addr(A[i][j]) = Base Address+ Offset= Base Address + (number of columns placed before jth column * size of column) * (Size of Element) + (number of elements placed before in ith element in ith row)* size of element***

- *For Element_Size = 1 the address is*
  - ***Address of A[i][j] for column major arrangement = Base + (j * m) + I***

- *In general, for column-major arrangement; address of the element of the jth row and the jth column therefore is*
  - ***Addr (A[i][j] = ((j – LB2) * (UB1 – LB1 + 1) * size) + ((i –LB1) * size)***

Example 2.1: *Consider an integer array, int A[3][4] in C++. If the base address is 1050, find the address of the element A[2][3] with row-major and column-major representation of the array.*

For C++, lower bound of index is 0 and we have m=3, n=4, and Base= 1050. Let us compute address of element A [2][3] using the address computation formula

1. Row-Major Representation:

*Address of A [2][3] = Base + (i * n) + j*

*= 1050 + (2 * 4) + 3*

= 1061

1	2	3	4	5	6	7	8	9	10	11	12
(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)	(2,0)	(2,1)	(2,2)	(2,3)

Row1 Row2 Row3

Row-Major Representation of 2-D array

2. Column-Major Representation:

*Address of A [2][3] = Base + (j * m) + i*

*= 1050 + (3 * 3) + 2*

= 1050 + 11

= 1061

- *Here the address of the element is same because it is the last member of last row and last column.*

1	2	3	4	5	6	7	8	9	10	11	12
(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)	(0,3)	(1,3)	(2,3)

Col 1 Col 2 Col 3 Col 4

Column-Major Representation of 2-D array

Characteristics of array

- An array is a finite ordered collection of homogeneous data elements.
- In array, successive elements of list are stored at a fixed distance apart.
- Array is defined as set of pairs-(index and value).
- Array allows random access to any element
- In array, insertion and deletion of element in between positions
- requires data movement.
- Array provides static allocation, which means space allocation done once during compile time, can not be changed run time.

Advantage of Array Data Structure

- Arrays permit efficient random access in constant time o(1).
- Arrays are most appropriate for storing a fixed amount of data and also for high frequency of data retrievals as data can be accessed directly.
- Wherever there is a direct mapping between the elements and there positions, arrays are the most suitable data structures.
- Ordered lists such as polynomials are most efficiently handled using arrays.
- Arrays are useful to form the basis for several more complex data structures, such as heaps, and hash tables and can be used to represent strings, stacks and queues.

Disadvantage of Array Data Structure

- *Arrays provide static memory management. Hence during execution the size can neither be grown nor shrunk.*
- *Array is inefficient when often data is to inserted or deleted as inserting and deleting an element in array needs a lot of data movement.*
- *Hence array is inefficient for the applications, which very often need insert and delete operations in between.*

Applications of Arrays

- *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.*
- *Two-dimensional data when represented as Matrix and matrix operations.*

CONCEPT OF ORDERED LIST

- *Ordered list is the most common and frequently used data object*
- *Linear elements of an ordered list are related with each other in a particular order or sequence*
- *Following are some examples of the ordered list.*

  - *1, 3,5,7,9,11,13,15*
  - *January, February, March, April, May, June, July, August, September,*
  - *October, November, December*
  - *Red, Blue, Green, Black, Yellow*

- ***There are many basic operations** that can be performed on the ordered list as follows:*

  - *Finding the length of the list*
  - *Traverse the list from left to right or from right to left*
  - *Access the ith element in the list*
  - *Update (Overwrite) the value of the ith position*
  - *Insert an element at the ith location*
  - *Delete an element at the ith position*

SINGLE VARIABLE POLYNOMIAL

2.8 Single Variable Polynomial

- A polynomial p(x) is the expression in variable x which is in the form $(ax^n + bx^{n-1} + \ldots + jx + k)$, where a, b, c ...., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.
- An important characteristics of polynomial is that each term in the polynomial expression consists of two parts:
  - One is the coefficient
  - Other is the exponent

Example

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 are its exponential value.

A polynomial of a single variable A(x) can be written as

$$a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots\ldots a_1 x + a_0 \quad \text{where } a_n \neq 0 \text{ and}$$

degree of A(x) is n

Single Variable Polynomial

- *Representation Using Arrays*
- *Array of Structures*
- *Polynomial Evaluation*
- *Polynomial Addition*
- *Multiplication of Two Polynomials*

Polynomial Representation on array

- It is possible to maintain polynomials such as $5x^4 - 10x^3 + 3x^2 + 10x - 1$ using array. On polynomial some basic operations such as addition and multiplication are often implemented. In such case there is need to represent polynomials.
- The easiest method of representing a polynomial having degree **n** is storing the coefficient of $(n + 1)$ terms of the polynomial in the respective array.
- For this purpose each and every array element must have two values; coefficient and exponent.
- In this representation, it is considered that the exponent of each successive term is less than its previous term.

Polynomial Representaiton

- Array representation of polynomial assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0.
- The coefficients of the respective exponent are placed at an appropriate index in the array.

For example

$Q(x) = 5x^4 - 10x^3 + 3x^2 + 10x - 1.$

$Q(x) = 5x^4 - 10x^3 + 3x^2 + 10x - 1$

Exponent	Coefficient
0	– 1
1	10
2	3
3	– 10
4	5
5	
6	

Fig. 2.8.1 : Polynomial Representation using Array

- *Polynomial as an ADT, the basic operations are as follows:*
  - *Creation of a polynomial*
  - *Addition of two polynomials*
  - *Subtraction of two polynomials*
  - *Multiplication of two polynomials*
  - *Polynomial evaluation*

Polynomial by using Array

POLYNOMIAL of degree 3 $P(x) = 3x^3+x^2-2x+5$						
INDEX i	0	1	2	3	.....	N-1
COEF	3	1	-2	0	....	0

POLYNOMIAL of degree 8 $P(x) = 11x^8+5x^6+ x^5+2x^4 -3x^2+ x +10$									
INDEX i	0	1	2	3	4	5	6	7	8
COEF	11	0	5	1	2	0	−3	1	10

Polynomial by using Array

POLYNOMIAL of degree 99 $P(x) = x^{99} + 78$						
INDEX i	0	1	2	3	.....	99
COEF	1	0	0	0	....	78

❖ Structure is better than array for Polynomial:

- *Such representation by an array is both time and space efficient when polynomial is not a sparse one such as polynomial P(x) of degree 3 where P(x)= 3x3+x2–2x+5.*
- *But when polynomial is sparse such as in worst case a polynomial as A(x)= x99 + 78 for degree of n =100, then only two locations out of 101 would be used.*
- *In such cases it is better to store polynomial as pairs of coefficient and exponent. We may go for two different arrays for each or a structure having two members as two arrays for each of coeff. and Exp or an array of structure that consists of two data members coefficient and exponent.*

Polynomial by using structure

- *Let us go for structure having two data members coefficient and exponent and its array.*

POLYNOMIAL of degree 99 P (x) = $3x^3+x^2-2x+5$						
INDEX i	0	1	2	3	.....	N-1
COEF	3	1	-2	5	....	
EXPO	3	2	1	0	.....	

SPARSE MATRIX

- *In many situations, matrix size is very large but out of it, most of the elements are zeros (not necessarily always zeros).*

- *And only a small fraction of the matrix is actually used. A matrix of such type is called a sparse matrix,*

2.9.1 Sparse Matrix Representation

Array Representation of Sparse Matrix

- 2D array is used to represent a sparse matrix in which there are three rows named as
- **Row** : Index of row, where non-zero element is located.
- **Column** : Index of column, where non-zero element is located

SPARSE MATRIX

– **Value :** Value of the non zero element located at index – (row, column)

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

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

Fig. 2.9.1 : Sparse Matrix

Let us take an example of a logical matrix LA and LB as follows:

$$LA = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}_{7 \times 5}$$

$$LB = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{pmatrix}_{7 \times 5}$$

Sparse Logical Matrix

Matrix A = $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 9 & 8 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 4 \\ 0 & 0 & 2 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$ 6 x 7

Alternate Representation of Sparse Matrix A=

6 Rows	7 Columns	8 NonZero Entries
1	1	1
2	2	9
2	3	8
3	1	3
4	3	5
4	4	4
5	2	2
5	3	3

9 x 3

Sparse matrix and its representation

Transpose Of Sparse Matrix

- *Simple Transpose*
- *Fast Transpose*

Transpose Of Sparse Matrix

– To transpose a matrix, we must interchange the rows and columns. This means that if an element is at position [j][k] in the original matrix, then it is at position [k][j] in the transposed matrix.

– When k = j, the elements on the diagonal will remain unchanged. Since the original matrix is organized by rows, our first idea for a transpose algorithm might be the following:

for (each row k)

take element (k,j, value) and

store it in (j,k, value) of the transpose;

For example

Sp1

Index	Row	Column	value
0	3	4	4
1	1	0	5
2	1	2	3
3	2	1	1
4	2	3	2

Fig. 2.9.2(a) : Spares Matrix

Sp2

Index	Row	Column	value
0	4	3	4
1	0	1	5
2	1	2	1
3	2	1	3
4	3	2	5

Fig. 2.9.2(b) : Transpose Matrix

Transpose Of Sparse Matrix

Step 1 : copy elements of column 0 of sp1 to sp2.

sp1

Row	Column	Value
3	4	4
1	$\boxed{0}$	5
1	2	3
2	1	1
2	3	2

element of column 0

sp2

Row	Column	Value
4	3	4
0	1	5

Step 2 : copy elements of column 1 of sp1 to sp2.

sp1

Row	Column	Value
3	4	4
1	0	5
1	2	3
2	$\boxed{1}$	1
2	3	2

element of column 0

sp2

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

Step 3 : copy elements of column 2 of sp1 to sp2.

sp1

Row	Column	Value
3	4	4
1	0	5
1	$\boxed{2}$	3
2	1	1
2	3	2

element of column 0

sp2

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

Time complexity of manual technique is O (mn).

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix} \qquad \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{pmatrix}$$

Sparse matrix transpose

3 x 4

Sparse Matrix and its Transpose

4 x 3

$$\text{Let } B = \left(\begin{array}{ccc} \mathbf{6} & \mathbf{7} & \mathbf{5} \\ \hline 1 & 2 & 7 \\ 2 & 4 & 2 \\ 3 & 6 & 5 \\ 5 & 0 & 4 \\ 5 & 3 & 9 \\ 6 & 1 & 8 \end{array}\right) \qquad B^T = \left(\begin{array}{ccc} \mathbf{7} & \mathbf{6} & \mathbf{5} \\ \hline 2 & 1 & 7 \\ 4 & 2 & 2 \\ 6 & 3 & 5 \\ 0 & 5 & 4 \\ 3 & 5 & 9 \\ 1 & 6 & 8 \end{array}\right)$$

Simple Sparse matrix transpose

- *Time complexity will be* $O\ (n.\ \boldsymbol{T})$

$$= \boldsymbol{O\ (n\ .\ mn)}$$

$$= \boldsymbol{O\ (mn^2)}$$

which is worst than *the conventional transpose with time complexity* $O\ (mn)$

Fast Sparse matrix transpose

- *In worst case, i.e. $T = m \times n$ (non-zero elements) the magnitude becomes $O(n + mn) = O(mn)$ which is the same as 2-D transpose*
- *However the constant factor associated with fast transpose is quite high*
- *When T is sufficiently small, compared to its maximum of m .* ***n, fast transpose will workfaster***

String Manipulation Using Array

- *It is usually formed from the character set of the programming language*
- *The value n is the length of the character string S where n* ^{3} *0*
- *If n = 0 then S is called a null string or empty string*

❖*Basically a string is stored as a sequence of characters in one- dimensional character array say A.*

char A[10] ="STRING" ;

Each string is terminated by a special character that is null character '\0'.

This null character indicates the end or termination of each string.

	0	1	2	3	4	5	6	7	8	9
A =	S	T	R	I	N	G	\0	-	-	-

- ***There are various operations that can be performed on the string:***

  - *To find the length of a string*
  - *To concatenate two strings*
  - *To copy a string*
  - *To reverse a string*
  - *String compare*
  - *Palindrome check*
  - *To recognize a sub string*

Write a program to reverse string.

```
#include<iostream>

#include<string.h>

using namespace std;

int main ()

{

   char str[50], temp;

   int i, j;

   cout << "Enter a string : ";

   Cin>>str;

   j = strlen(str) - 1;

   for (i = 0; i < j; i++,j--)

   {

      temp = str[i];

      str[i] = str[j];

      str[j] = temp;

   }

   cout << "\nReverse string : " << str;

   return 0;

}
```

/* C++ Program - Concatenate String using inbuilt function */

```
#include<iostream.h>
#include<string.h>
void main()
{
clrscr();
char str1[50], str2[50];
cout<<"Enter first string : ";
Cin>>str1;
cout<<"Enter second string : ";
 cin>>str2;
strcat(str1, str2);
cout<<"String after concatenation is "<<str1;
getch();
}
```

```
/* C++ Program - Concatenate String without using inbuilt function */

#include <iostream>
using namespace std;
int main()
 {
 char str1[100] = "Hi...";
char str2[100] = "How are you";
int i,j;
cout<<"String 1: "<<str1<<endl;
cout<<"String 2: "<<str2<<endl;
for(i = 0; str1[i] != '\0'; ++i);
j=0;
while(str2[j] != '\0')
{
str1[i] = str2[j];
i++;      j++;
 }
str1[i] = '\0';
cout<<"String after concatenation: "<<str1;
return 0;
}
```

To get the length of a C-string string, strlen() function is used.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
char str[] = "C++ Programming is awesome";
// you can also use str.length()
cout << "String Length = " << strlen(str);
return 0;
}
```

Find length of string without using strlen function

```
#include<iostream>
using namespace std;
int main()
{
char str[] = "Apple";
int count = 0;
while (str[count] != '\0')
count++;
cout<<"The string is "<<str<<endl; cout <<"The length of the string is "<<count<<endl;
return 0;
}
```

/* C++ Program - Compare Two String */

```
#include<iostream.h>
#include<string.h>
void main()
 {
 clrscr();
char str1[100], str2[100];
cout<<"Enter first string : ";
gets(str1);
cout<<"Enter second string : ";
gets(str2);
if(strcmp(str1, str2)==0)
{
cout<<"Both the strings are equal";
 }
else
{ cout<<"Both the strings are not equal";
 } getch(); }
```

String is palindrom or not

```
#include <iostream>
#include <string.h>
using namespace std;
int main()
{
char str1[20], str2[20];
int i, j, len = 0, flag = 0;
cout << "Enter the string : ";
gets(str1);
len = strlen(str1) - 1;
for (i = len, j = 0; i >= 0 ; i--, j++)
str2[j] = str1[i];
if (strcmp(str1, str2))
flag = 1;
if (flag == 1)
cout << str1 << " is not a palindrome";
else
cout << str1 << " is a palindrome";
return 0;
}
```

SIMPLE TRANSPOSE OF MATRIX IN C++

```
#include <iostream>

using namespace std;

int main()

{

int a[10][10], trans[10][10], r, c, i, j;

cout << "Enter rows and columns of matrix: ";

cin >> r >> c;// Storing element of matrix

cout << endl << "Enter elements of matrix: " <<
endl;

for(i = 0; i < r; ++i)

for(j = 0; j < c; ++j)

{

cout << "Enter elements a" << i + 1 << j + 1 << ":
";

cin >> a[i][j];

} // Displaying the matrix a[][]

cout << endl << "Entered Matrix: " << endl;

for(i = 0; i < r; ++i)

for(j = 0; j < c; ++j)
{ cout << " " << a[i][j];

if(j == c - 1)

cout << endl << endl; }

// Finding transpose of matrix a[][] and storing it
infor(i = 0; i < r; ++i)

for(j = 0; j < c; ++j)

{

trans[j][i]=a[i][j];

}

// Displaying the transpose

cout << endl << "Transpose of Matrix: " << endl;

for(i = 0; i < c; ++i)

for(j = 0; j < r; ++j)

{

cout << " " << trans[i][j];

if(j == r - 1)

cout << endl << endl;

}

return 0;  }
```

MULTIPLICATION OF TWO POLYNOMIAL

```
#include<math.h>
#include<stdio.h>
#include<conio.h>
#define MAX 17
void init(int p[]);
void read(int p[]);
void print(int p[]);
void add(int p1[],int p2[],int p3[]);
void multiply(int p1[],int p2[],int p3[]);

/*Polynomial is stored in an array, p[i] gives coefficient
of x^i .
  a polynomial 3x^2 + 12x^4 will be represented as
(0,0,3,0,12,0,0,….)
*/

void main()
{
int p1[MAX],p2[MAX],p3[MAX];
int option;
do
{
printf("nn1 : create 1'st polynomial");
printf("n2 : create 2'nd polynomial");
printf("n3 : Add polynomials");
printf("n4 : Multiply polynomials");
printf("n5 : Quit");
printf("nEnter your choice :");
scanf("%d",&option);
switch(option)
{
case 1:read(p1);break;
case 2:read(p2);break;
case 3:add(p1,p2,p3);
      printf("n1'st polynomial -> ");
      print(p1);
      printf("n2'nd polynomial -> ");
      print(p2);
      printf("n Sum = ");
      print(p3);
      break;
case 4:multiply(p1,p2,p3);
      printf("n1'st polynomial -> ");
      print(p1);
      printf("n2'nd polynomial -> ");
      print(p2);
      printf("n Product = ");
      print(p3);
      break;
}
}while(option!=5);
}
```

MULTIPLICATION OF TWO POLYNOMIAL

```
void read(int p[])
{
int n, i, power,coeff;
init(p);
printf("n Enter number of terms :");
scanf("%d",&n);
/* read n terms */
for (i=0;i<n;i++)
{      printf("nenter a term(power  coeff.)");
scanf("%d%d",&power,&coeff);
p[power]=coeff;
}
}
void print(int p[])
{
int i;
for(i=0;i<MAX;i++)
if(p[i]!=0)
printf("%dX^%d   ",p[i],i);
}
void add(int p1[], int p2[], int p3[])
{
   int i;
   for(i=0;i<MAX;i++)
p3[i]=p1[i]+p2[i];
}
```

```
void  multiply(int p1[], int p2[], int p3[])
{
int i,j;
init(p3);
for(i=0;i<MAX;i++)
for(j=0;j<MAX;j++)
   p3[i+j]=p3[i+j]+p1[i]*p2[j];
}
void init(int p[])
{
    int i;
    for(i=0;i<MAX;i++)
p[i]=0;
}
```

********END*******

FIND SIMPLE TRANSPOSE OF MATRIX

```
#include <iostream>
using namespace std;
int main()
{
   int a[10][10], trans[10][10], r, c, i, j;

   cout << "Enter rows and columns of matrix: ";
   cin >> r >> c;

//Storing element of matrix enter by user in array a[][].
   cout << endl << "Enter elements of matrix: " << endl;
   for(i = 0; i < r; ++i)
   for(j = 0; j < c; ++j)
   {
      cout << "Enter elements a" << i + 1 << j + 1 << ": ";
      cin >> a[i][j];
   }
// Displaying the matrix a[][]
   cout << endl << "Entered Matrix: " << endl;
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
      {
         cout << " " << a[i][j];
         if(j == c - 1)
            cout << endl << endl;
      }

   // Finding transpose of matrix a[][] and storing it in array trans[][].
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
      {
         trans[j][i]=a[i][j];
      }
```

FIND SIMPLE TRANSPOSE OF MATRIX

```
// Displaying the transpose,i.e, Displaying array trans[][].
   cout << endl << "Transpose of Matrix: " << endl;
   for(i = 0; i < c; ++i)
      for(j = 0; j < r; ++j)
      {
         cout << " " << trans[i][j];
         if(j == r - 1)
            cout << endl << endl;
      }

   return 0;
}
```

THANK YOU !!!!!

Blog : anandgharu.wordpress.com

gharu.anand@gmail.com