UNIT- II
Linear Data Structures Using Sequential Organization

Contents

- Sequential Organization, Linear Data Structure Using Sequential Organization, Array as an Abstract Data Type, Memory Representation and Address Calculation, Inserting an element into an array, Deleting an element, Multidimensional Arrays, Two-dimensional arrays, n-dimensional arrays, Concept of Ordered List, Single Variable Polynomial, Representation using arrays, Polynomial as array of structure, Polynomial addition, Polynomial multiplication, Sparse Matrix, Sparse matrix representation, Sparse matrix addition, Transpose of sparse matrix, String Manipulation Using Array,
- Case Study- Use of sparse matrix in Social Networks and Maps.

Introduction

- We have already studied that there are multiple ways to organize data (Unit 1).
- Data organization heavily affects programming logic.
- We therefore select **data structures and algorithms** in such a way that the overall program proves to be efficient in terms of **space and time complexities**.

Linear data structure

- The term “linear” means “belonging to a line”
- A Linear data structure consists of elements which are ordered i.e. in a line.
- The elements form a sequence such that there is a first element, second, …… and last element.
- Thus, the elements of a linear data structure have a one-one relationship.
- Examples : array, list.

- When each element may have one or more successors (or predecessors), it is called a non-linear data structure.
- Link List is Linear data structure

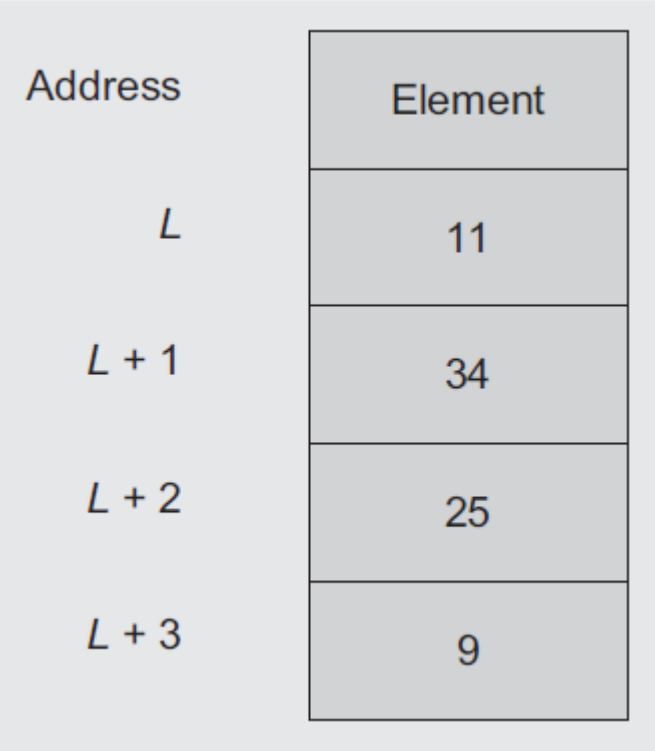
General Problem-Solving Concepts

Sequential Organization

- As the name suggests, sequential organization allows storing data a fixed distance apart.
- If the i^{th} element is stored at location X, then the next sequential $(i + 1)^{th}$ element is stored at location $X + C$, where C is a constant.

Sequential Organization

- One major advantage of sequential organization is the **direct or random** access to any data element of the list in constant time.
- As sequential organization uses **continuous memory locations** to store data, the data access time remains constant for accessing any element of the list, irrespective of the total length or size of the data list.



Linear Data Structure Using Sequential Organization: Arrays

- To store a group of data together in a sequential manner in computer's memory, arrays can be one of the possible data structures.
- Arrays enable us to organize more than one element in consecutive memory locations; hence, it is also termed as *structured* or *composite data type*.
- The only restriction is that all the elements we wish to store must be of the same data type.

Linear Data Structure Using Sequential Organization: Arrays

- Arrays are the most general and easy to use of all the data structures.
- 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
- Index allows the direct addressing (or accessing) of any element of an array.

Array

- An *array* is a **finite ordered collection** of **homogeneous** data elements that provides direct access to any of its elements.
- Arrays can be used in any of their varied forms. A one-dimensional array is the simplest form of an array. Each word in the definition has a specific meaning:
- ***Finite*** The number of elements in an array is finite or limited.
- **Ordered collection** The arrangement of all the elements in an array is very specific, that is, every element has a particular ranking in the array.
- ***Homogeneous*** All the elements of an array should be of the same data type.
- Let us see how to declare an array in C++.
- int Array_A[20];

Array

- The common terms associated with arrays are as follows:
- **Size of array** The maximum number of elements that would be stored in an array is the size of that array.
- **Base** The base address of an array is the memory location where the first element of an array is stored.
- **Data type of an array** The data type of an array indicates the data type of elements stored in that array.

Array

- **Index** A user or a programmer can access the elements of an array by using subscripts such as Name[0], Name[1], ..., Name[i].
- **Range of index** If *N* is the size of an array, then in C++, the range of index is 0 - (*N* - 1)

Array as an ADT

- An array is a set of pairs, index and value.
- For each index, there exists one associated element of an array.
- For defining an array as an abstract data type (ADT), we have to define the very basic operations or functions that can be performed on it.
- The basic operations of arrays are:
  - Creating an array
  - Storing an element
  - Accessing an element
  - Traversing the array

Array as an ADT

- **Methods:**

for all A ∈ Array, i ∈ index, x ∈ item, j, size ∈ integer

Array Create(j, list) ::= **return** an array of j dimensions where list is a j-tuple whose kth element is the size of the kth dimension. Items are undefined.

Item Retrieve(A, i) ::= **if** (i ∈ index) **return** the item associated with index value i in array A
else return error

Array Store(A, i, x) ::= **if (**i in index)
return an array that is identical to array A except the new pair <i, x> has been inserted **else return** error

Memory Representation And Address Calculation

0	1	2	3	4
10	20	30	40	50
2000	2004	2008	2012	2016
A[0]	A[1]	A[2]	A[3]	A[4]

- The lower bound of array starts at 0 and upper bound is (size-1)
- The operation store is implemented as :
  - arrayname[position] = value ;
  - e.g. A[2] = 30;
- The operation extract is implemented as :
  - arrayname[position]
  - e.g. num = A[2];

Memory Representation And Address Calculation

0	1	2	3	4
10	20	30	40	50
2000	2004	2008	2012	2016
A[0]	A[1]	A[2]	A[3]	A[4]

- Address of A[i] = Base Address + i * element size
- E.g.
- Address of A[3] = 2000 + 3 * sizeof(int)

= 2000 + 3 * 2

= 2006

Operations on Arrays

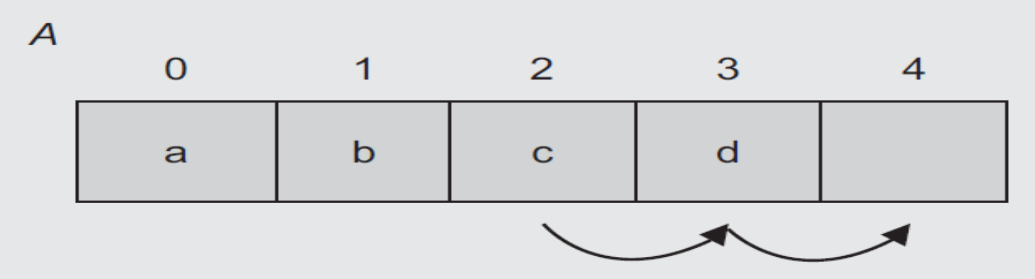
- Inserting an element into an array
- Deleting an element from the array
- Inserting an element into an sorted array
- Merging two sorted arrays

Inserting an element into an array

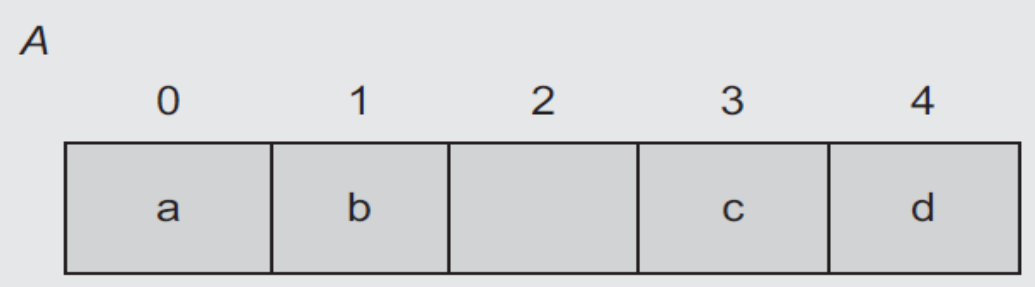
- To insert an element at the i^{th} position in an array of size N, all the elements originally at positions i, $i + 1$, $i + 2$, ..., $N – 1$ will be shifted to $i + 1$, $i + 2$, $i + 3$, ..., N, respectively so that each element gets shifted to the right by one position.
- All the data shifting must be performed before the actual insertion.

Inserting an element into an array

Consider the following array:

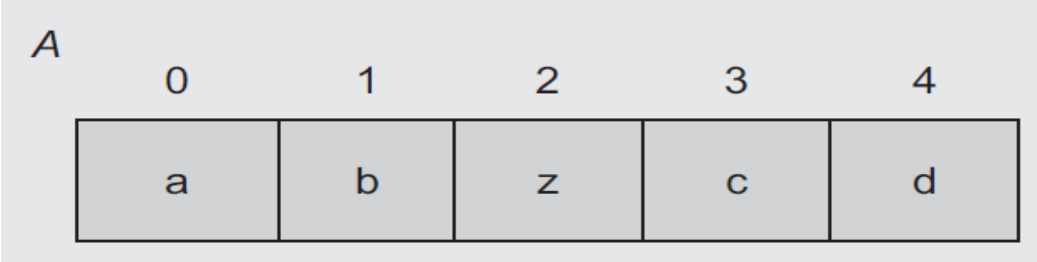


To insert ‘z’ at index = 2, that is at position 3, create room at 3 by data shifting.



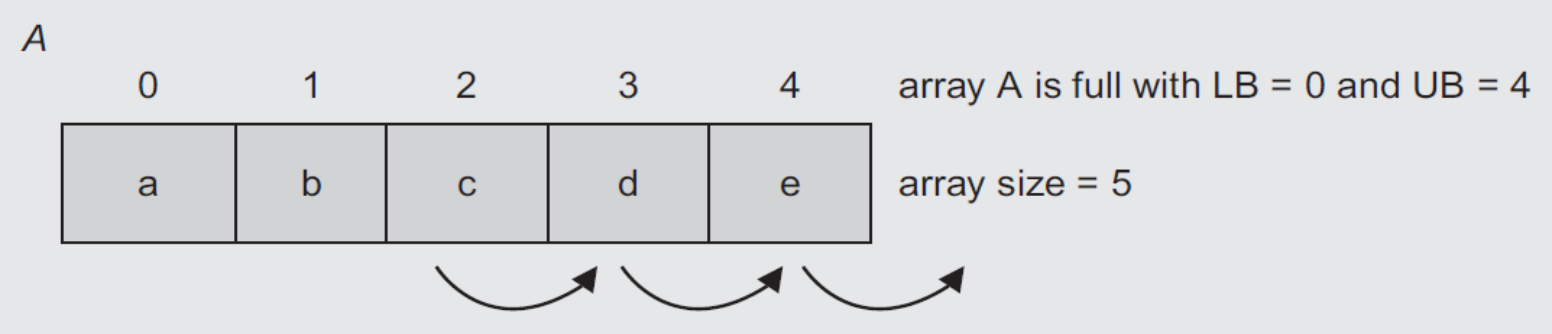
Inserting an element into an array

Then insert ‘z’ at position 3.



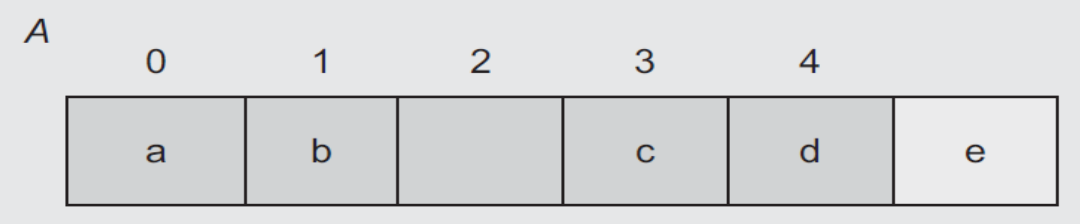
Inserting an element into an array

- If the array is already full before the insertion of a new element, the last element of the array will be lost after insertion because of array overflow.
- Now, consider the following array *A*:

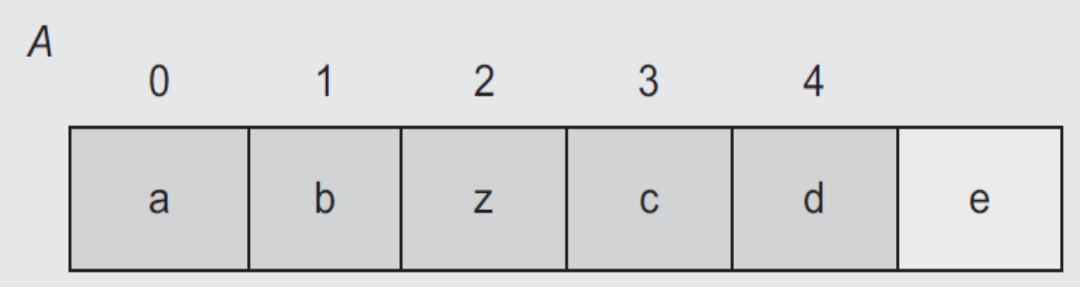


Inserting an element into an array

To insert ‘z’ at position 3, create room at the 3rd position by data shifting.



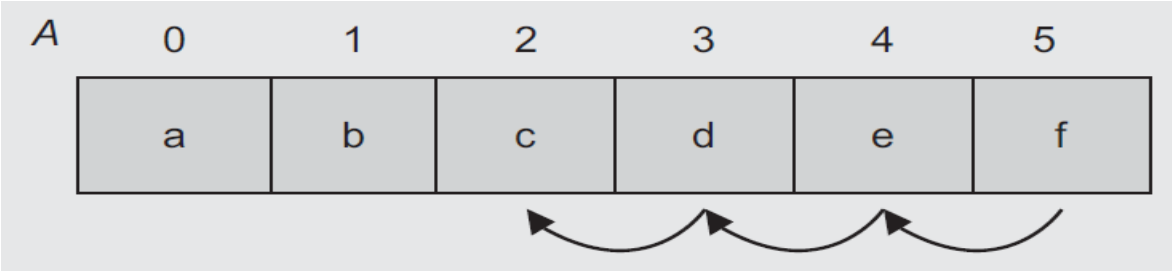
Then insert ‘z’ at position 3.



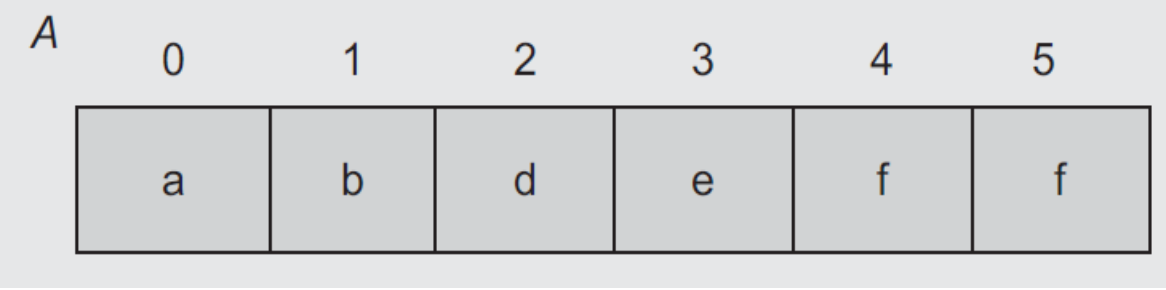
As the element ‘e’ is shifted to index 5, ‘e’ becomes inaccessible

Deleting an element from an array

- Deletion of an element is achieved by overwriting the element. After one deletion operation, one location becomes empty, so all the elements should be shifted by one position after the deleted element to fill in the empty location of the deleted element.

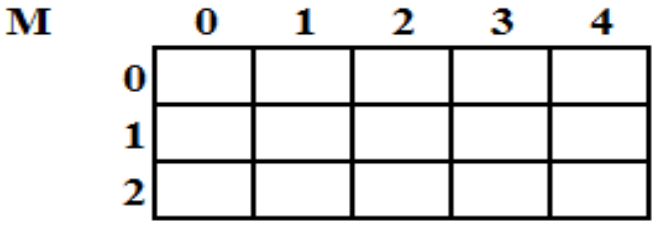


Delete ‘c’ from the 3rd position, that is, index = 2.



Multidimensional Arrays

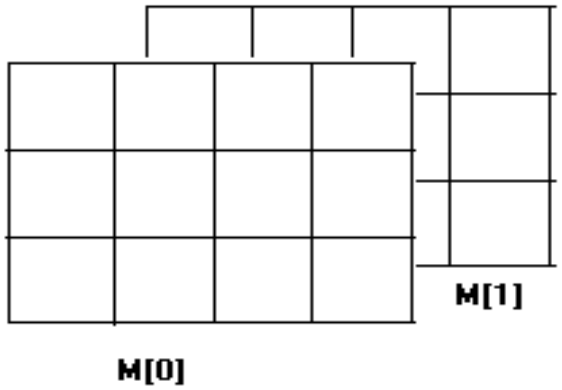
- When each array element is another array, it is called a multidimensional array.
- Example:
  - int M[3][5];
  - declares M as an array containing 3 elements each of which is an array containing 5 elements.



- An element of this array is accessed by specifying two indices. Such an array is called a two-dimensional array.
- M is said to have 3 rows and 5 columns. The rows are numbered 0 to 2 and columns are numbered 0 to 4.

Multidimensional Arrays

- An array can have more than two dimensions. For example, a three dimensional array may be declared as :
  - int M[2][3][4];
  - M is an array containing 2 two-dimensional arrays, each having 3 rows and 4 columns.



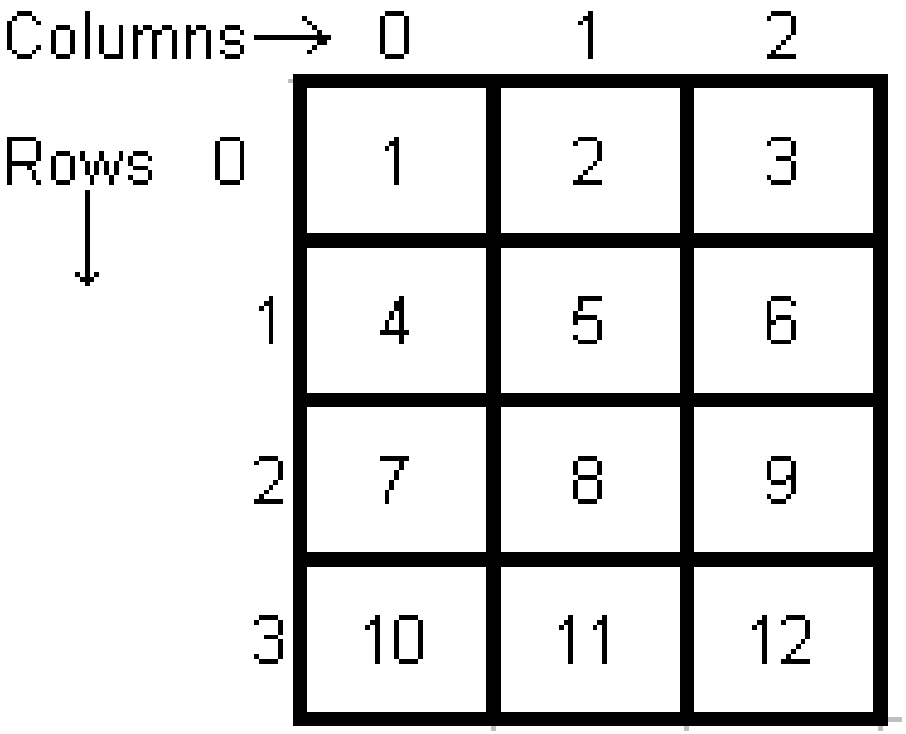
Representation of Multidimensional arrays

- A two-dimensional array is a logical data structure that is useful in describing an object that is physically two-dimensional such as a matrix or a check board. This is the logical view of data.
- However, computer memory is linear, i.e. it is essentially a one-dimensional array. Thus, the elements of a two-dimensional array have to be stored linearly in memory.

Representation of Multidimensional arrays

- They can be represented in two ways.
  - **Row-major representation :** In this representation, the elements are arranged row-wise i.e. the 0^{th} row elements are stored first, the next row elements after them and so on.
  - **Column-major representation :** Elements are arranged columnwise i.e. all elements of the 0th column occupy the first set of memory locations, elements of the first column come next and so on.

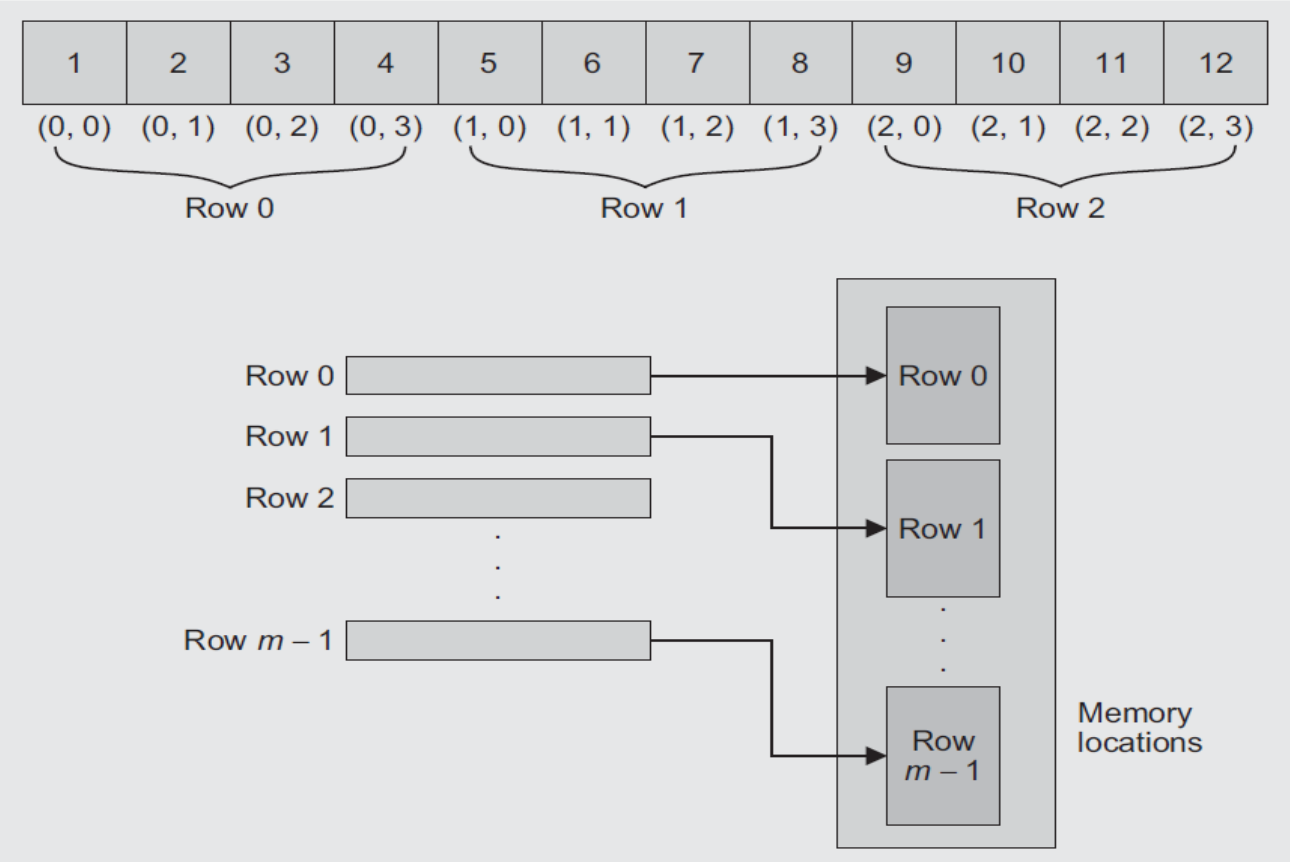
Logical View



Row-major Representation

- In row-major representation the elements of matrix *M* are stored row-wise, that is, elements of the 0^{th} row, 1^{st} row, 2^{nd} row, 3^{rd} row, and so on till the m^{th} row.

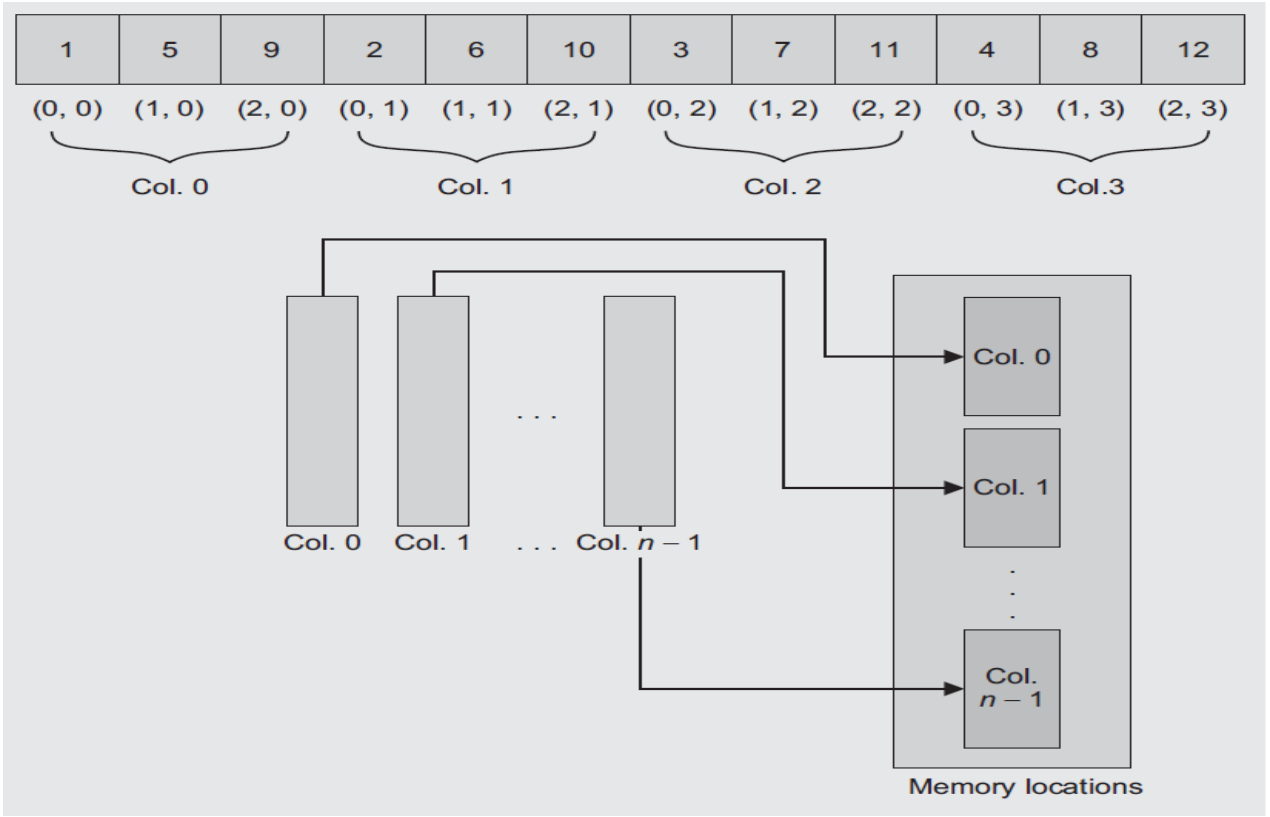
$$M = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}_{3 \times 4}$$



Column-major Representation

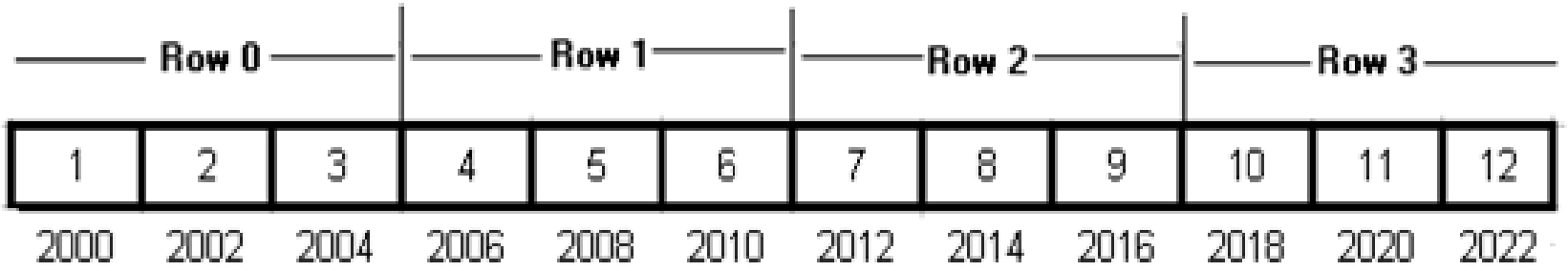
- The elements are stored in the memory as a sequence: first the elements of column 0, then the elements of olumn 1, and so on, till the elements of column *n* - 1.

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

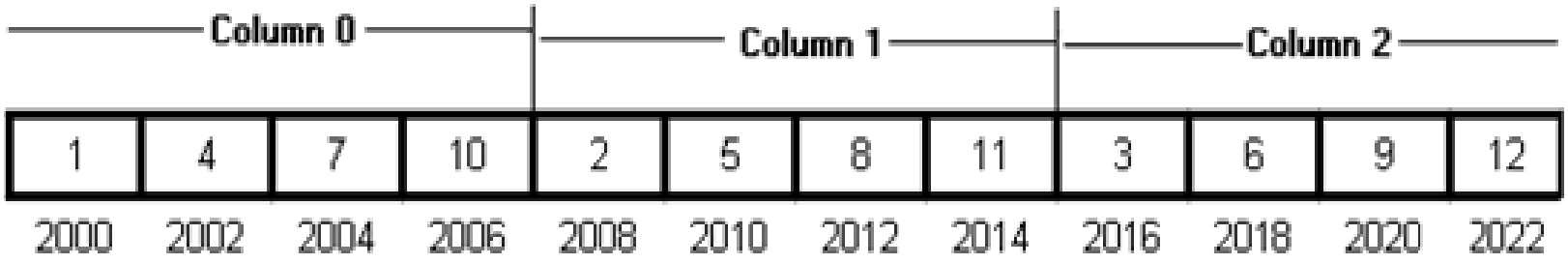


Physical View

Physical View : Row Major Representation

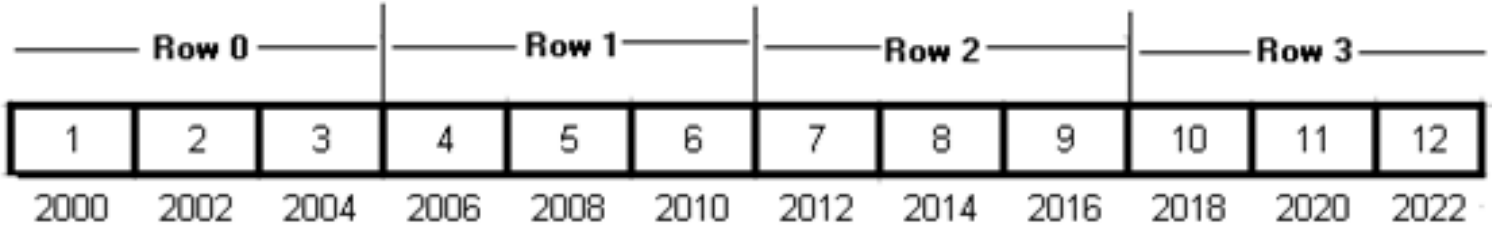


Physical View : Column Major Representation



Address Calculation

Physical View : Row Major Representation



For an array int M[R][C]; where R = number of rows and C = number of columns , the location of an element M[i][j] in the array can be calculated as :

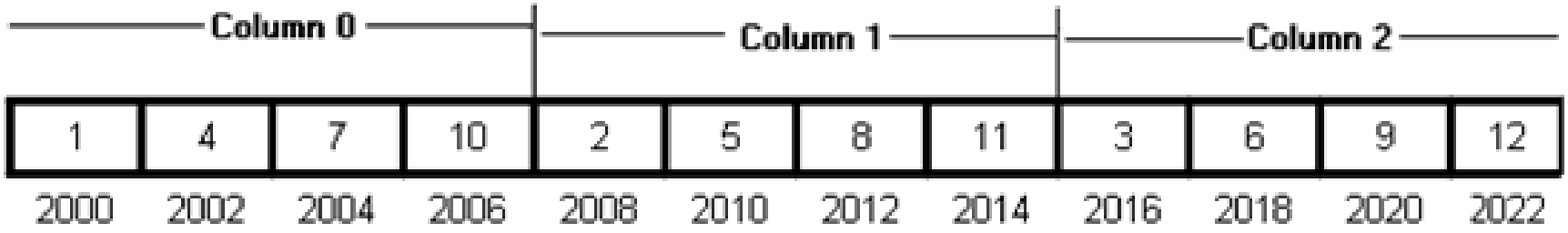
Address of M[i][j] = Base Address + i * C * Element size + j * Element size
= Base Address + (i * C + j) * Element size

Example :

Address of M[1][2] = 2000 + (1 * 3 + 2) * sizeof (int)
= 2000 + 5 * 2
= 2010

Address Calculation

Physical View : Column Major Representation



For an array int M[R][C]; where R = number of rows and C = number of columns , the location of an element M[i][j] in the array can be calculated as :

Address of M[i][j] = Base Address + j * R * Element size + i * Element size
= Base Address + (j * R + i) * Element size

Example :

Address of M[1][2] = 2000 + (2 * 4 + 1) * sizeof (int)
= 2000 + 9 * 2
= 2018

Concept of Ordered List

- Ordered List is nothing but a set of elements. Such a list sometimes called as linear list.
- **Definition :** An ordered list is set of elements where set may be empty or it can be written as a collection of elements such as (a1,a2,a3,……,an) .
- Example :
  - Days of the week
    { Mon, Tue, Wed, Thu, Fri, Sat, Sun }
  - Months of the year
    { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}
  - Values on the card deck
    { 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A }
  - List of one digit number
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Ordered List

- **Operations on ordered list**
  1. Finding the length of the list.
  2. Retrieving the i^{th} element from the list.
  3. Display of list
  4. Searching a particular element from the list.
  5. Insertion of any element in the list.
  6. Deletion of any element from the list.

- **Applications of Ordered List**
  - Polynomial Representation
  - Stacks
  - Queues, etc.

Ordered List

- Arrays are the most common data structures that can be used for representing an ordered list.
- In an ordered list, members of the list follow some specific sequence.
- The best possible way to organize them is in an array. Let L be the list; $L = \{a_0, a_1, a_2, ..., a_{n-1}\}$ having n elements.

list[0]	list[1]	list[2]	. . .	list[n – 1]
a_0	a_1	a_2	. . .	a_{n-1}

The representation of an ordered list L in array

Single Variable Polynomial

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

  $a_n x_n + a_{n-1} x_{n-1} + a_{n-2} x_{n-2} + \ldots \ldots a_1 x + a_0$

  where $a_n \neq 0$ and degree of $A(x)$ is n.
- This polynomial is a sum of terms $C.x^e$ where C is a coefficient, e is the exponent, and x is a variable.
- An example of a polynomial of a single variable x is $x^2 - 4x + 7$.

Single Variable Polynomial

- A polynomial is one of the examples of an ordered list.
- When we think of a polynomial as an ADT, the basic operations are as follows:

1. Creation of a polynomial
2. Addition of two polynomials
3. Subtraction of two polynomials
4. Multiplication of two polynomials
5. Polynomial evaluation

Representation using arrays

- One Dimensional Array
- Two Dimensional Array
- Array of Structures

Representation using 1-D arrays

- $F(x) = 2x^4 + x^2 - 4x + 2$,
- int P[10], n; n is the no. of terms in the polynomial.
- Coefficient index : 2i
- Exponent index : 2i + 1

0	1	2	3	4	5	6	7
2	4	1	2	-4	1	2	0

Representation using 2-D arrays

- $F(x) = 2x^4 + x^2 - 4x + 2$,
- int P[10][2], n; //n is the no. of terms in the polynomial.
- Coefficient : P[i][0]
- Exponent : P[i][1]

Coef Exp

	0	1	
0	2	4	1st Term
1	1	2	2nd Term
2	-4	1	3rd Term
3	2	0	4th Term

Representation using array of str

- $F(x) = 2x^4 + x^2 - 4x + 2$,

```
typedef struct MYPOLY
{
    int coef;
    int exp;
}mypoly;
```

- mypoly P[10] ,n; //n is the no. of terms in the polynomial.
- Coefficient : P[i].coef Exponent : P[i].exp

```
struct polynomial{
    int coef;
    int expo;
};
class poly{
    public:
        polynomial poly_array[100];
        int total_terms;
        poly()
        {
            total_terms = 0;
        }
        void accept();
        void display();
        double evaluate(double);
        poly operator +(poly);
        poly operator *(poly);
};
```

```
void poly :: accept(){
    cout<<"Enter Total Number Of Terms : ";
    cin>>total_terms;
    cout<<endl;
    for(int i=0;i<total_terms;i++){
        cout<<"Enter Coefficient Of "<<i+1<<" Term :
";
        cin>>poly_array[i].coef;
        cout<<"Enter Power Of "<<i+1<<" Term : ";
        cin>>poly_array[i].expo;
        cout<<endl;
    }
}

void poly :: display(){
    for(int i=0;i<total_terms;i++){

cout<<poly_array[i].coef<<"x^"<<poly_array[i
].expo<<" ";
    }
    cout<<endl;
}
```

Operations on Polynomials

- **Evaluating a polynomial**: For the given value of x, figure out what value you get.
- **Evaluate $2x^3 - x^2 - 4x + 2$ at $x = -3$**
- $F(-3) = 2(-3)^3 - (-3)^2 - 4(-3) + 2$

  $= 2(-27) - (9) + 12 + 2$

  $= -54 - 9 + 14$

  $= -63 + 14$

  $= \mathbf{-49}$

```
double poly :: evaluate(double value){
    int i = total_terms;
    double result = 0;
    while(i--){
        result += poly_array[i].coef*pow(value, poly_array[i].expo);
    }
    return result;
}
```

Operations on Polynomials

- **Adding Polynomials :** To add two polynomials, simply combine like terms.
- Egs : $(5x^2 + 6x - 3) + (2x^2 - 7x - 9) = 7x^2 - x - 12$

- **Multiplying Polynomials :** To multiply two polynomials multiply **each term** in one polynomial by **each term** in the other polynomial, add those answers together, and simplify if needed.
- Egs : $(x + 2)(x^2 + 4x - 7) = x^3 + 6x^2 + x - 14$

Adding Polynomials

- Let two polynomials A and B be
- $A = 4x^9 + 8x^6 + 5x^3 + x^2 + 4x$
- $B = 3x^7 + x^3 - 2x + 5$
- Then,
- $C = A + B = 4x^9 + 3x^7 + 8x^6 + 6x^3 + x^2 + 2x + 5$

Adding Polynomials

- Let **i, j, and k be the three indices** to keep track of the current term of the polynomials A, B, and C, respectively
- If the exponent of the term indicated by **i in A is less than the exponent of the current term specified by j of B**, then copy the current term of B pointed by j in the location pointed by k in polynomial C. The pointers j and k are advanced to the next term.
- If the exponent of the term pointed **by j in B is less than the exponent of the current term pointed by i of A**, then copy the current term of A pointed by i in the location pointed by k in polynomial C. Advance the pointer i and k to the next term.

$$A = 4x^9 + 8x^6 + 5x^3 + x^2 + 4x$$

index i	0	1	2	3	4
Coefficient	4	8	5	1	4
Exponent	9	6	3	2	1

$$B = 3x^7 + x^3 - 2x + 5$$

index j	0	1	2	3
Coefficient	3	1	-2	5
Exponent	7	3	1	0

$$C = 4x^9 + 3x^7 + 8x^6 + 6x^3 + x^2 + 2x + 5$$

index k	0	1	2	3	4	5	6
Coefficient	4	3	8	6	1	2	5
Exponent	9	7	6	3	2	1	0

1. Read two polynomials say A and B
2. Let M and N denote total terms in A and B respectively.

Here, C is resultant polynomial.

4. Let i = j = k = 0
5. while (i < M and j < N) do
   begin // repeat till one of the polynomials is copied
   if(A[i].Exp = B[j].Exp)
   begin
   C[k].Coef = A[i].Coef+B[j].Coef
   C[k].Exp = A[i].Exp;
   i = i + 1; j = j + 1, k = k + 1
   end
   else
   if(A[i].Exp > B[j].Exp)
   begin
   C[k].Coef = A[i].Coef;
   C[k].Exp = A[i].Exp;
   i = i + 1
   k = k + 1
   end
   else
   begin
   C[k].Coef = B[j].Coef;
   C[k].Exp = B[j].Exp;
   j = j + 1
   k = k + 1
   end
   end
   6. while(i < m) do
   begin // copy remaining terms
   C[k].Coef = A[i].Coef;
   C[k].Exp = A[i].Exp;
   i = i + 1
   k = k + 1
   end
   7. while (j < n) do
   begin // copy remaining terms
   C[k].Coef = B[j].Coef;
   C[k].Exp = B[j].Exp;
   j = j + 1
   k = k + 1
   end

8) stop

Operations on Polynomials

- **Multiplying Polynomials : ** To multiply two polynomials multiply **each term** in one polynomial by **each term** in the other polynomial, add those answers together, and simplify if needed.
- Egs : $(x + 2)(x^2 + 4x - 7) = x^3 + 6x^2 + x - 14$

Multiplying Polynomials

Let A = $4x^9 + 3x^6 + 5x^3 + 1$, B = $3x^6 + x^2$, and C be the resultant polynomial. Initially, C is an empty polynomial.

1. We multiply each term of A with the first term of B. To start with, multiply $4x^9$ with $3x^6$ and the result is $12x^{15}$. Currently, C is empty, so there is no term in it with the exponent 15; therefore, we insert it in polynomial C. Now, polynomial C is

$$C = 12x^{15}$$

Multiplying Polynomials

Now, continue to multiply $3x^6$ with $3x^6$, and the result obtained is $9x^{12}$. There is no term in polynomial C with exponent 12, so we insert it in polynomial C at an appropriate location. Now, polynomial C is

$$C = 12x^{15} + 9x^{12}$$

Continuing in a similar manner for the remaining two terms of polynomial A, we get polynomial C as

$$C = 12x^{15} + 9x^{12} + 15x^9 + 3x^6$$

Multiplying Polynomials

Now, multiply each term of A with the second term of B. Initially, multiply $4x^9$ with x^2 and the result is $4x^{11}$. There is no term in C with exponent 11, we insert it in polynomial *C* at an appropriate location. So now we get polynomial C as

$$C = 12x^{15} + 9x^{12} + 4x^{11} + 15x^9 + 3x^6$$

Multiplying Polynomials

Continue to multiply $3x^6$ with x^2 and the result is $3x^8$. There is no term in polynomial C with exponent 8, so we add it at an appropriate place. Now, the polynomial C is

$$C = 12x^{15} + 9x^{12} + 4x^{11} + 15x^9 + 3x^8 + 3x^6$$

Let us now multiply $5x^3$ with x^2 and we get $5x^5$. There is no term in C with exponent 5, so we insert it in polynomial C at a proper location. Now,

$$C = 12x^{15} + 9x^{12} + 4x^{11} + 15x^9 + 3x^8 + 3x^6 + 5x^5$$

Multiplying Polynomials

Let us now multiply the term 1 of A with x^2; we get x^2. There is no term in C with exponent 2, so we insert it in polynomial C at an appropriate location. Therefore,

$$C = 12x^{15} + 9x^{12} + 4x^{11} + 15x^9 + 3x^8 + 3x^6 + 5x^5 + x^2$$

This is the resultant polynomial C as a result of A * B.

Multiplying Polynomials

1. Let A and B be two polynomials.

2. Let the number of terms in A be M, and number of terms in B be N.

3. Let C be the resultant polynomial to be computed as C = A * B.

4. Let us denote the *i*th term of polynomial B as *t*B*i*. For each term *t*B*i* of polynomial B, repeat steps 5 to 7 where *i* = 1 to N.

5. Let us denote the *j*th term of polynomial A as *t*A*j*. For each term of *t*A*j* of polynomial A, repeat steps 6 and 7 where *j* = 1 to M.

6. Multiply *t*A*j* and *t*B*i*. Let the new term be *t*C*k* = *t*A*j* * *t*B*i*.

7. Compare *t*C*k* with each term of polynomial C. If a term with equal exponent is found, then add the new term *t*C*k* to that term of polynomial C, else search for an appropriate position for the term *t*C*k* and insert the same in polynomial C.

8. Stop.

Sparse matrix

- A matrix is a very commonly used mathematical object.
- To represent a matrix, we need a two-dimensional array with two different indices for row and column references.
- The representation of a matrix for operations on it should be efficient so that the space and time requirement is less.
- In many situations, the matrix size is very large but most of the elements in it are zero's
- A matrix of such type is called a *sparse matrix*.

Sparse matrix

- In such cases, the matrix must be represented and stored with an alternate representation to achieve good space utilization.

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

The matrix ***LA*** is sparse with respect to 1s and dense with respect to 0s, whereas ***LB*** is sparse with respect to 0s and dense with respect to 1s.

Sparse matrix

- For a matrix of *m* rows and *n* columns, if *m* = *n*, then the matrix is called a square matrix.

$$A = \begin{pmatrix} 0 & 0 & 6 & 1 & 1 & 2 \\ 0 & 1 & 0 & 0 & 9 & 9 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 6 \\ 0 & 2 & 0 & 0 & 0 & 8 \end{pmatrix}_{6 \times 6}$$

The matrix ***A*** in has many 0 entries, and it may be called a sparse matrix.

Sparse matrix

- Two general types of *n*–square sparse matrices are
  - Sparse triangular matrix
  - Sparse tridiagonal matrix

A matrix in which all non-zero entries occur only on or below the main diagonal is called **a *triangular matrix***.

A matrix in which the non-zero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a ***tridiagonal matrix***

$$B = \begin{pmatrix} 10 & 0 & 0 \\ 21 & 90 & 0 \\ 45 & 28 & 15 \end{pmatrix}_{3\times 3}$$

$$C = \begin{pmatrix} 9 & 88 & 0 & 0 \\ 22 & 8 & 95 & 0 \\ 0 & 33 & 6 & 44 \\ 0 & 0 & 56 & 47 \end{pmatrix}_{4\times 4}$$

Representation of Sparse Matrix

- A Sparse matrix can be stored as a list of tuples storing only the non-zero elements and their row and column number.
- The first tuple gives the information about the matrix namely number of rows, number of columns and the number of non-zero elements. The remaining give information about each non-zero value.
- Data Structure
  - 2-D Array
  - Array of Stucture

Row No	Column No	Value
5	**8**	**6**
0	0	10
0	4	-5
1	2	1
2	4	-4
3	0	7
4	1	9

Efficient representation of Sparse Matrix

- Consider the matrix ***M*** . Among the 40 elements, 6 members are nonzero.
- For conventional representation, we need 40 memory locations for storing the matrix (assuming one location per element), whereas for its alternate representation as in (b), we need (6 + 1) * 3, that is, 21 memory locations.

M =

$$\begin{bmatrix} 10 & 0 & 0 & 0 & -5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Row No	Column No	Value
5	**8**	**6**
0	0	10
0	4	-5
1	2	1
2	4	-4
3	0	7
4	1	9

Addition of Sparse Matrix

Sparse Matrix A

Row No	Column No	Value
5	**8**	**6**
0	0	10
0	4	-5
1	2	1
2	4	-4
3	0	7
4	1	9

+

Sparse Matrix B

Row No	Column No	Value
5	**8**	**5**
0	1	2
0	4	7
1	1	3
1	2	2
3	0	4

Addition Result

Row No	Column No	Value
5	**8**	**8**
0	0	10
0	1	2
0	4	2
1	1	3
1	2	3
2	4	-4
3	0	11
4	1	9

Transpose a Matrix

- In the conventional approach, by interchanging rows and columns, we get the transpose of the matrix as the elements at position [i][j] and [j][i] are swapped.
- Let *m* and *n* be the number of rows and columns for matrix ***A***. The transpose of ***A*** can be obtained using the following code.

```
for(i = 1; i ≤ m; i++)
        for(j = 1; j ≤ n; j++)
                A[j][i] = A[i][j];
```

- Time complexity of this technique is $O(mn)$.

- The matrix B is a simple sparse matrix of size 6 X 7 with 5 non-zero elements and its transpose.
- We can notice that entries in B^T are not sorted row and column wise; we need to sort them further.
- Sorting further adds to time complexity. Let us learn two better approaches the simple and fast transpose algorithms.

$$B = \begin{pmatrix} \mathbf{6} & \mathbf{7} & \mathbf{5} \\ 1 & 2 & 7 \\ 2 & 4 & 2 \\ 3 & 6 & 5 \\ 5 & 0 & 4 \\ 5 & 3 & 9 \\ 6 & 1 & 8 \end{pmatrix} \qquad B^T = \begin{pmatrix} \mathbf{7} & \mathbf{6} & \mathbf{5} \\ 2 & 1 & 7 \\ 4 & 2 & 2 \\ 6 & 3 & 5 \\ 0 & 5 & 4 \\ 3 & 5 & 9 \\ 1 & 6 & 8 \end{pmatrix}$$

Simple Transpose of a Sparse Matrix

- Let A be a matrix of size *m* X *n* with T non-zero elements and let B be its transpose. One of the easiest ways is to search for each column (column = 0 to *n* - 1) and sequentially place each column as a row in the transposed matrix B by placing the interchanged entries as row, column, and value

Simple Transpose of a Sparse Matrix

1. Row = A[0][0], Col = A[0][1] and T = A[0][2]

2. B[0][0] = Col , B[0][1] = Row and B[0][2] = T

3. if T = 0 goto step(5)

4. Let i = 1

for j = 0 to Col-1 do

    for k = 1 to T do

    if(A[k][1] = j)

    begin

    B[i][0] = A[i][1]
    B[i][1] = A[i][0]
    B[i][2] = A[i][2]

    i = i + 1

    end

5. stop

Analysis of Simple Transpose

- If there are n columns in matrix A of order m x n and the number of non-zero terms is t, the computing time is O(nt).
- If t is the order of nm, the time for this algorithm becomes $O(n^2m)$ which is far worse than O(nm) time for transpose of normal matrices.

Fast Transpose of a Sparse Matrix

- This method achieves the transpose of a sparse matrix in O(n+t) time and hence is much faster than the simple transpose method. This method is as follows
  - Find the number of elements in each column of A
  - This gives the number of elements in each row of T
  - Using this, calculate the starting point of each row of T.
  - Move each element of A one by one to its correct position in T.

Example : Fast Transpose

Row No	Column No	Value
5	8	6
0	0	10
0	4	-5
1	2	1
2	4	-4
3	0	7
4	1	9

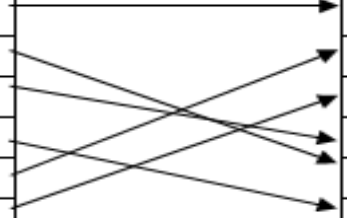
Number of elements in each column of A

	0	1	2	3	4	5	6	7
Count	2	1	1	0	2	0	0	0

The starting positions of each row in T.

	0	1	2	3	4	5	6	7
Position	0	2	3	4	4	6	6	6

Row No	Column No	Value
5	8	6
0	0	10
0	4	-5
1	2	1
2	4	-4
3	0	7
4	1	9



Row No	Column No	Value
5	8	6
0	0	10
0	3	7
1	4	9
2	1	1
4	0	-5
4	2	-4

Fast Transpose of a Sparse Matrix

```
void Fast_transpose_sparse_matrix(struct sparse s[],struct sparse t[])
{
   static int pos[15],term[15];
   t[0].row = s[0].col;
   t[0].col = s[0].row;
   t[0].val = s[0].val;
   for(i=1;i<=s[0].val;i++)
         term[s[i].col]++;
   pos[0] = 1;
   for(i=1;i<s[0].col;i++)
      pos[i] = pos[i-1] + term[i-1];
   for(i=1;i<=s[0].val;i++)
   {
       t[pos[s[i].col]].row = s[i].col;
       t[pos[s[i].col]].col = s[i].row;
       t[pos[s[i].col]].val = s[i].val;
       pos[s[i].col]++;
    }
}
```

Analysis of Fast Transpose

- The steps taken to find number of elements in each column of A = n [Assuming a m x n matrix]
- The steps taken to compute the starting positions of each row in T = n -1
- The elements of A are copied to T in t steps
- Therefore, Total no. of steps = n + n – 1 + t
- Which is of the order O(n+t).
- If t is of the order nm, this method becomes O(nm) which is same for two dimensional arrays.

Characteristics of Array

The characteristics of an array are as follows:

1. An array is a finite ordered collection of homogeneous data elements.
2. In an array, successive elements are stored at a fixed distance apart.
3. An array is defined as a set of pairs—index and value.
4. An array allows direct access to any element.
5. In an array, insertion and deletion of elements in-between positions require data movement.
6. An array provides static allocation, which means the space allocation done once during the compile time cannot be changed during run-time.

Advantages of Array

The various merits of the array as a data structure are as follows:

1. Arrays permit efficient random access in constant time 0(1).

2. 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.

3. 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 (unlike a linked list in which each data element has an additional link field).

Advantages of Array

4. Arrays are well known in applications such as searching, hash tables, matrix operations, and sorting.

5. Wherever there is a direct mapping between the elements and their position, such as an ordered list, arrays are the most suitable data structures.

6. Ordered lists such as polynomials are most efficiently handled using arrays.

7. Arrays are useful to form the basis for several complex data structures such as heaps and hash tables and can be used to represent strings, stacks, and queues.

Disadvantages of Array

Some of the disadvantages of arrays are as follows:

1. Arrays provide static memory management. Hence, during execution, the size can neither be grown nor shrunk.

2. There is a solution to handle the problem, that is, to declare the array of some arbitrarily maximum size. This leads to two other problems:

(a) In future, if the user still needs to exceed this limit, it is not possible.

b) Higher the maximum, the more is the memory wastage because very often, many locations remain unused but still allocated (reserved) for the program. This leads to poor utilization of space.

3. Static allocation in an array is a problem associated with implementation in many programming languages except a few such as JAVA.

Disadvantages of Array

4. An array is inefficient when often data is inserted or deleted as insertion or deletion of an element in an array needs a lot of data movement.

5. Hence, an array is inefficient for the applications that often need insert and delete operations in between.

6. A drawback due to the simplicity of arrays is the possibility of referencing a nonexistent 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. In the worst case, the whole system can crash. In C++, the powerful syntax is unfortunately prone to this kind of error. Some languages have built-in bounds checking and do not index an array outside of its permitted range.

String Manipulation Using Array

- A string is a character array terminated by a null character (\0)
- Operations on Strings
  - String Length
  - String Copy
  - String Concatenation
  - String Reverse
  - String Compare
  - Substring,etc

Thank You