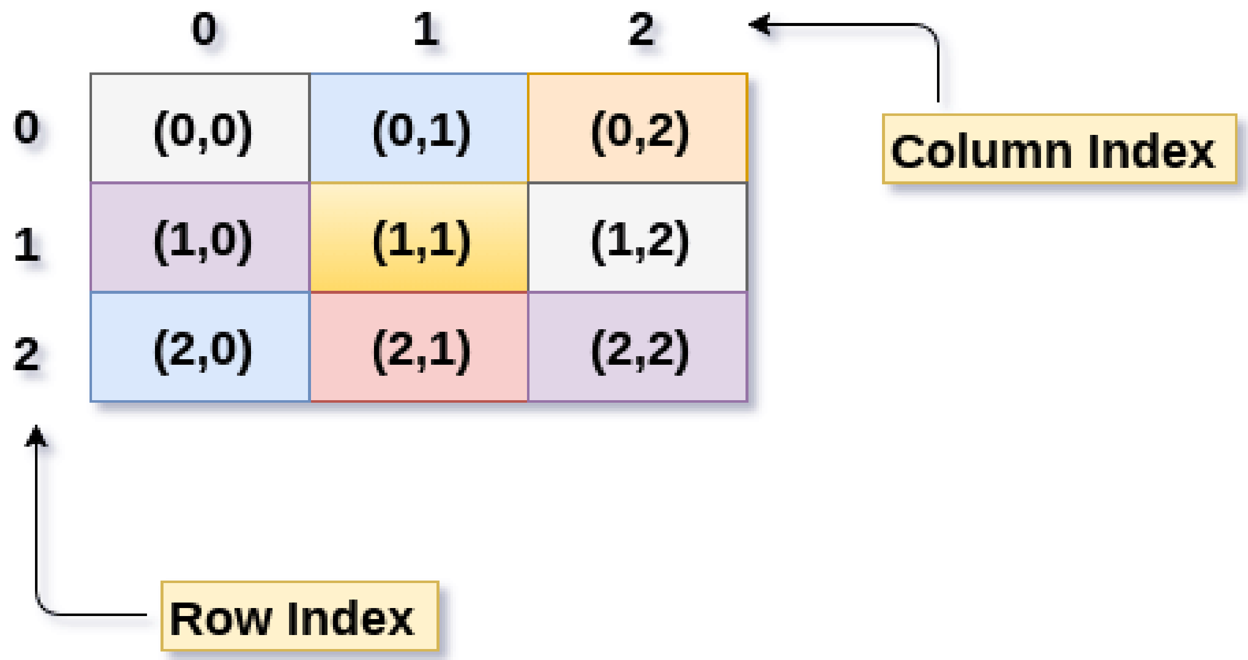
2-Dimension Arrays

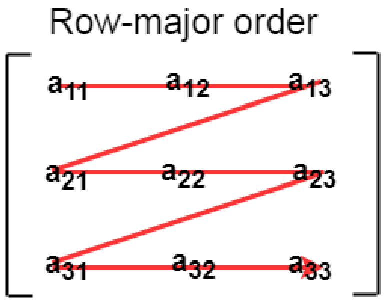
2 D Arrays



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

1. Row Major ordering

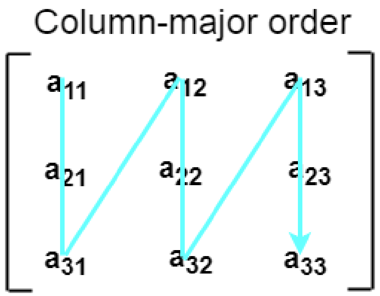
- In **row major ordering**, all the **rows** of the 2D array are **stored into the memory contiguously**.
- Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.



(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)

2. Column Major ordering

- According to the **column major ordering**, all the **columns** of the 2D array **are stored into the memory contiguously**.
- The memory allocation of the array which is shown in in the above image is given as follows.



(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)

Row Major Order

- In row major order, the elements of a particular row are stored at adjacent memory locations.
- The first element of the array (arr[0][0]) is stored at the first location followed by the arr[0][1] and so on.
- After the first row, elements of the next row are stored next.
- arr[3][3] =
  [a00, a01, a02]
  [b10, b11, b12]
  [c20, c21, c22]
- Row major order = **a00, a01, a02, b10, b11, b12, c20, c12, c22**

If the first element is stored at memory location 1048 and the elements are integers, then

[1048] - a00
[1052] - a01
[1056] - a02
[1060] - b10
[1064] - b11
[1068] - b12
[1072] - c20
[1076] - c21
[1080] - c22

Column Major Order

- In column major order, the elements of a **column are stored adjacent to each other** in the memory.
- The first element of the array (arr[0][0]) is stored at the first location followed by the arr[1][0] and so on.
- After the first column, elements of the next column are stored stating from the top.
- arr[3][3] =
  [a00, a01, a02]
  [b10, b11, b12]
  [c20, c21, c22]
- Column major order = a00, b10, c20, a01, b11, c21, a02, b12, c22

If the first element is stored at memory location 1048 and the elements are integers, then:

[1048] - a00
[1052] - b10
[1056] - c20
[1060] - a01
[1064] - b11
[1068] - c21
[1072] - a02
[1076] - b12
[1080] - c22

```
#include <stdio.h>
#include <time.h>
int m[999][999];
//Taking both dimensions same so that while running the loops, number of operations (comparisons, iterations, initializations)
//are exactly the same. Refer this for more  https://www.geeksforgeeks.org/a-nested-loop-puzzle/
 void main()
 {
    int i, j;
    clock_t start, stop;
    double d = 0.0;
     start = clock();
    for (i = 0; i < 999; i++)
        for (j = 0; j < 999; j++)
            m[i][j] = m[i][j] + (m[i][j] * m[i][j]);
     stop = clock();
    d = (double)(stop - start) / CLOCKS_PER_SEC;    /* CLOCKS_PER_SEC defines the number of clock ticks per second for a particular machine.
                                                      It is a MACRO defined in time.h   */
    printf("The run-time of row major order is %lf\n", d);
     start = clock();
    for (j = 0; j < 999; j++)
        for (i = 0; i < 999; i++)
            m[i][j] = m[i][j] + (m[i][j] * m[i][j]);
     stop = clock();
    d = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("The run-time of column major order is %lf", d);
}
```

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

s[0][0]	s[1][0]	s[2][0]	s[3][0]	s[0][1]	s[1][1]	s[2][1]	s[3][1]
1234	**1212**	**1434**	**1312**	**56**	**33**	**80**	**78**
65508	65510	65512	65514	65516	65518	65520	65520

Assumption: sizeof(int) = 2 Bytes

Example – 2D Array

ROW MAJOR

```
#include<stdio.h>
void main()
{
int a[4][2]= {1234,56,1212,33,1434,80,1312,78}, i, j;
for(i=0;i<4;i++)
{
   for(j=0;j<2;j++)
    {
        printf("%d\t", a[i][j]);
     }
     printf("\n");
}
}
```

```
1234    56
1212    33
1434    80
1312    78
```

COLUMN MAJOR

```
#include<stdio.h>
void main()
{
int a[4][2]= {1234,56,1212,33,1434,80,1312,78}, i, j;
for(j=0;j<2;j++)
{
      for(i=0;i<4;i++)
       {
        printf("%d\t", a[i][j]);
        }
     printf("\n");
}
}
```

```
1234    1212    1434    1312
56      33      80      78
```

Address Mapping in Arrays

To calculate the individual array element address:

Let, **B be the base address of an array and**

<u>S be the size of each element</u>

1 D Array:

The location of i th element

$= B + (i – 1) * S$

2 D Array:

Row Major:

The location of a[i][j] element

$= B + S * (i * \mathbf{n} + j)$

Where **n** is total no of columns

Column Major:

The location of a[i][j] element

$= B + S * (j * m + i)$

Where **m** is total no of rows

Example – 1 D Array

a

12	34	66	-45	23	346	77	90
65508	65510	65512	65514	65516	65518	65520	65522

Let, **B be the base address of an array and**
S be the size of each element

```
#include<stdio.h>
void main()
{
int a[8]= {12,34,66,-45,23,346,77,90}, i ;
for(i=0;i<8;i++)
{
print(“%d\t”,&a[i]);
}
}
```

Printing the Address of each element of Array

OUTPUT:
65508 65510 65512 65514 65516 65518 65520 65522

1 D Array:

The location of i th element

$= B + (i – 1) * S$

e.g:

Address of 4 th element in a

$= 65508 + (4 – 1) * 2$

$= \mathbf{65514}$

Example – 2D Array

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

ROW MAJOR

```
#include<stdio.h>
void main()
{
int a[4][2]= {1234,56,1212,33,1434,80,1312,78}, i, j;
for(i=0;i<4;i++)
{
   for(j=0;j<=2;j++)
    {
     print(“%d\t”,&a[i][j]);
     }
}
}
```

OUTPUT:

65508 65510 65512 65514 65516 65518 65520 65522

Let, **B be the base address of an array and**
S be the size of each element
And n is the total number of elements per row

Row Major:

The location of s[i][j] element

$$= B + S * (i * n + j)$$

Where **n** is total no of columns

Therefore:

Address of s[1][0] = 65508 + 2 * (1 * **2** + 0)

= 65512

Example – 2D Array

COLUMN MAJOR

s[0][0]	s[1][0]	s[2][0]	s[3][0]	s[0][1]	s[1][1]	s[2][1]	s[3][1]
1234	**1212**	**1434**	**1312**	**56**	**33**	**80**	**78**
65508	65510	65512	65514	65516	65518	65520	65520

```
#include<stdio.h>
void main()
{
int s[4][2]= {1234,56,1212,33,1434,80,1312,78}, i, j;
for(j=0;j<2;j++)
{
   for(i=0;i<4;i++)
    {
     print("%d\t",&a[i][j]);
     }
}
}
```

OUTPUT:

65508 65510 65512 65514 65516 65518 65520 65520

Let, **B be the base address of an array and**
S be the size of each element
And n is the total number of elements per row

Column Major:

The location of a[i][j] element

$= B + S * (j * m + i)$

Where **m** is total no of rows

Where **n** is total no of columns

Therefore:

Address of s[1][0] = 65508 + 2 * (0 * **4** + 1)

= 65510

MATRIX

- A **matrix** is a two-dimensional data object made of m rows and n columns, therefore having total m x n values.
- If most of the elements of the matrix have 0 value, then it is called a **sparse matrix**.

Sparse Matrix

- A matrix is sparse if many of its elements are zero
- A matrix that is not sparse is dense
- The boundary is not precisely defined
  - Diagonal and tridiagonal matrices are sparse
  - We classify triangular matrices as dense

Why to use Sparse Matrix instead of simple matrix ?

- **Storage**: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time**: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

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

- Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases.
- So, instead of storing zeroes with non-zero elements, **we only store non-zero elements.**
- This means storing non-zero elements with triples- (**Row, Column, value**).

Sparse Matrix

- Two possible representations
  - array
  - linked list

Method 1: Using Arrays:

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

Array Representation of Sparse Matrix

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

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

Time Complexity: O(NxM)
where N is the number of rows in the sparse matrix, and
M is the number of columns in the sparse matrix.

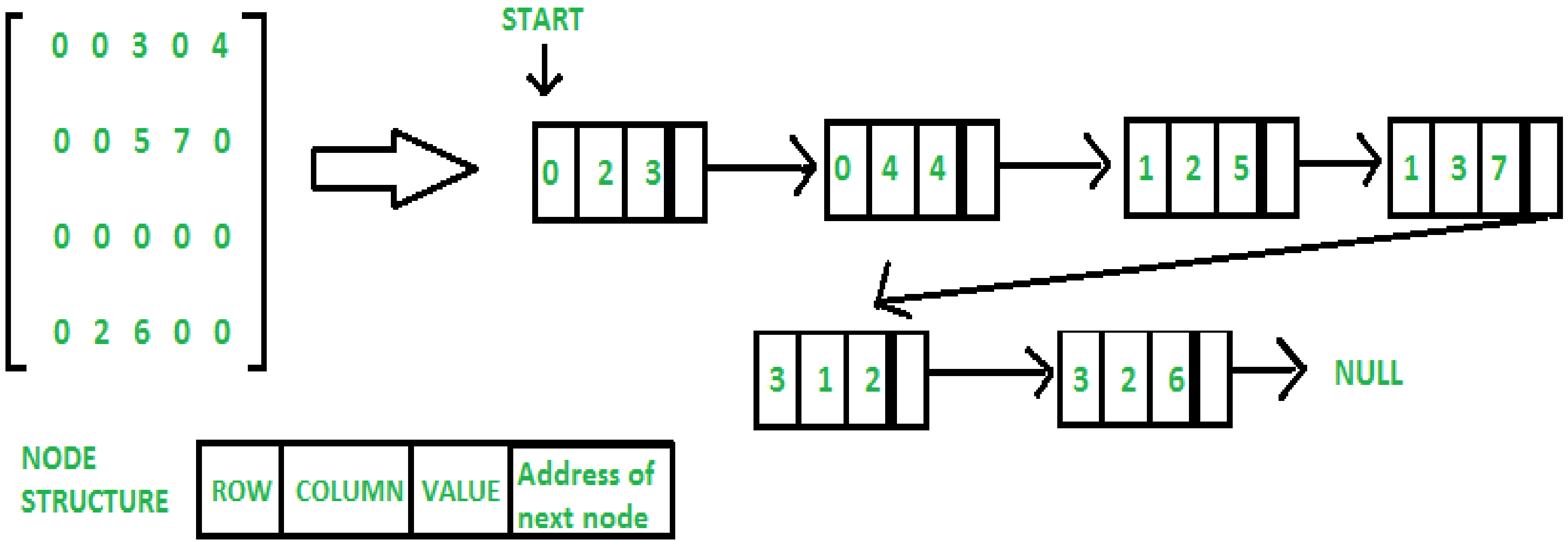
Method 2: Using Linked Lists

In linked list, each node has four fields.

These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node

Method 2: Using Linked Lists



Method 2: Using Linked Lists

```
// Creating head/first node of list as NULL
struct Node *first = NULL;
 struct Node *temp , *p;
for(int i = 0; i < 4; i++)
{
   for(int j = 0; j < 4; j++)
   {
       // Pass only those values which are non - zero
      if (sparseMatrix[i][j] != 0)
              { //Create a Node
                temp = (struct Node *)malloc(sizeof(struct Node));
                temp->row = row_index;
                temp->col = col_index;
                 temp->data = x;
                 temp->next = NULL;
              // If link list is empty then attach newly created node as first node
              if (first== NULL)
              {
              first = temp;
              }
    // If link list is already created then append newly created node
              else
               {      p = first;
                            while (p->next != NULL)
                            {          p = p->next; }

                            p->next = temp;
              }
              } // End of  j – For Loop
              } // End of i– For Loop
```

int sparseMatrix[4][5] = { { 0 , 0 , 3 , 0 , 4 },
{ 0 , 0 , 5 , 7 , 0 },
{ 0 , 0 , 0 , 0 , 0 },
{ 0 , 2 , 6 , 0 , 0 } };

Time Complexity: O(N*M), where N is the number of rows in the sparse matrix, and M is the number of columns in the sparse matrix.