Arrays

Array overview

- Collection of homogeneous data elements stored at contiguous memory location

For 1D array:

- Let **ARR[lb,ub]** be an array where **lb** and **ub** are values of lowest index and largest index .
- then size of array (n) =ub – lb + 1

- To access an element in array, we write

name_array[subscript] like ARR[i]
OR
name_array(subscript) like ARR(i)
OR
$name\_array_{subscript}$ like ARR_i

Memory representation of 1D Array

- Address of first element of an array **a** is known as base address – **base(a)**
- Address of k^{th} element is given by
  - **loc(a[k]) = base(a) + w * (k - lb)** where w is number of bytes per element
  - w can determined by data types of the element.
  - E.g. for int w=4 , for double w=8, for char w=1, and for long double w = 10
- If lb=0 (C ,C++ has array indexing 0 to n-1)
  - **loc(a[k]) = base(a) + w * k**

- **Note: To access any element in array constant time is needed because other elements scanning is not required.**

2D array (Matrices)

- Two subscripts are needed to access any element.

  Let **m** and **n** are the number of rows and columns in array then,

  Size of 2D array = m x n

- To access an element in array, we write

  name_array[rsubscript][csubscript] like ARR[i][j] // C, C++ style

  OR

  name_array(rsubscript, csubscript) like ARR(i][j)

  OR

  name_array$_{\text{rsubscript csubscript}}$ like ARR$_{i,j}$

  OR

  name_array[rsubscript, csubscript] like ARR[i,j] used in PASCAL

Memory representation of 2D Array

- Recall that address of first element of array **a** is known as base address – **base(a)**

Row major order:

- Let **w** be number of bytes per element, **lbc** is lower bound of columns and **lbr** is lower bound of rows.
- Address of element of i^{th} row and j^{th} column an array of mxn size is given by,

loc(a[i] [j]) = base(a) + w * [n*(i - lbr) + (j - lbc)]

loc(a[i] [j]) = base(a) + w * [n* i + j] In C/C++

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

Let base(a) = 2013
loc[2][1] = 2013 + 4* [4 * 2 + 1]
= 2013 + 4 * 9
= 2049

NOTE: C/C++ uses row major order for 2D arrays.

Memory representation of 2D Array (Contd…)

Column major order:

- Let **w** be number of bytes per element, **lbc** is lower bound of columns and **lbr** is lower bound of rows.
- So, ij element address in array of mxn is given by,

loc(a[i] [j]) = base(a) + w * [m*(j - lbc) + (i - lbr)]

loc(a[i] [j]) = base(a) + w * [m* j + i] In C/C++

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

Let base(a) = 2013
loc[2][1] = 2013 + 4 * [5 * 1 + 2]
= 2013 + 4 * 7
= 2041

NOTE: Pascal, Fortran and Matlab use column major order for 2D arrays.

Problem

double percentage[50];

If this array **percentage** is stored in memory starting from address **3000**. then what will be the address of **percentage[20]** element.

Ans: 3000+20*8= 3160

What will be size of array in bytes?

Ans:400

Operations on arrays

Traversing – Visiting each element exactly once. Complexity is O(n)

Insertion – Insert an element at given position. Complexity is O(n)

Deletion – Remove an element from given position. Complexity is O(n)

Searching – 1. Linear Search 2. Binary Search

Sorting – Arrange the elements of array on some logical order

1. Bubble sort 2. Insertion sort 3. Selection sort

4. Quick sort 5. Merge sort

Operations on arrays

Traversing – Traversing an array A of size N.

Traverse(A,N)

1. **Set i=0**
2. **While(i<N)**
3. **Print A[i]**
4. **i=i+1**
5. **STOP**

Time complexity in best case =O(n)

Time complexity in worst case =O(n)

Operations on arrays

Insertion – Insert an element P at index location k .

Insert(A[N],k,P)

1. **Set i=N**
2. **While(i>k)**
3. **A[i]=A[i-1] //shift toward right**
4. **i=i-1**
5. **A[k]=P**
6. **N=N+1**
7. **STOP**

Time complexity in best case =O(1) (When k is N i.e. insert at last)

Time complexity in worst case =O(n) (When k is 0 i.e. insert at first)

Operations on arrays

Deletion – Remove an element from index location k.

Delete(A[N],k)

1. **Set D=A[k]**
2. **i=k**
3. **While(i<=N-2)**
4. **A[i]=A[i+1] //shift left**
5. **i=i+1**
6. **N=N-1**
7. **STOP**

- Time complexity in best case =O(1) (When k is N-1 i.e. delete last element)
- Time complexity in worst case =O(n) (When k is 0 i.e. delete first element)

Sparse Matrix

- Most of the elements are zero
- If not sparse then it is called dense matrix
- As many of its elements are zero, we can save space by storing only non-zero elements.
- A sparse matrix can be represented by using following TWO representations:

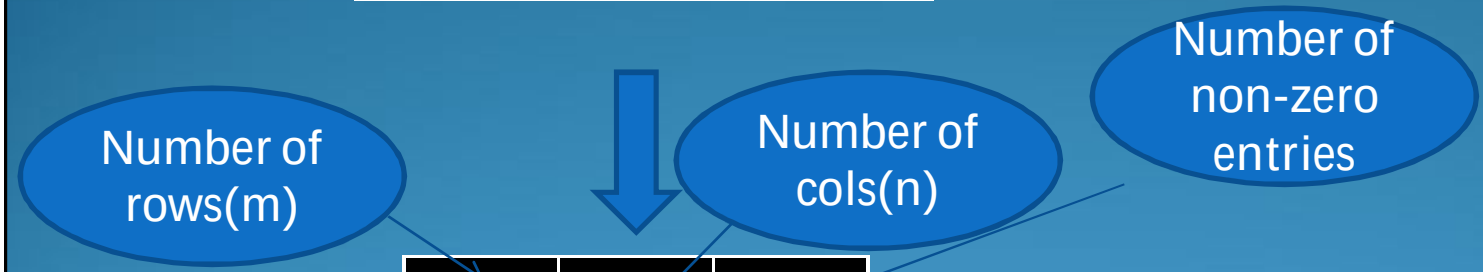
1. Triplet Representation or Array representation
2.Linked Representation (To be discussed later)

Sparse Matrix representation using array

Example:

0	2	0	0
0	1	0	0
3	0	0	0
0	0	0	0

Number of non-zero entries = 3
Number of zero entries = 13
Elements to be stored = 16



4	4	3
1	2	2
2	2	1
3	1	3

Only non-zero elements:

<row, col, element>

Elements to be stored = 12

Let see one more example –

Number of non-zero entries = 16
Number of zero entries = 26
Elements to be stored = 42

0	5	0	9	0	0
0	45	0	0	8	0
8	0	0	0	0	9
67	0	0	5	5	0
0	5	0	30	0	0
8	0	0	8	0	0
8	0	0	7	0	31

Number of non-zero entries = k
Number of rows = m
Number of columns = n
Condition: **3(k+1) < m*n**

Elements = 51

7	6	16
1	2	5
1	4	9
2	2	45
2	5	8
3	1	8
3	6	9
4	1	67
4	4	5
4	5	5
5	2	5
5	4	30
6	1	8
6	4	8
7	1	8
7	4	7
7	6	31

Advantages of Sparse Matrices

- Two advantages:
  - **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.

Is this Sparse Matrix? If YES then represent it.

0	0	0	0	3	0
0	9	0	0	0	2
0	0	1	8	0	0
0	0	0	0	4	0
3	0	0	0	0	0

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

YES

Here

Number of non-zero entries = k=7

Number of rows = m=5

Number of columns = n=6

Condition: **3(k+1) < m*n=3(7+1)<30**