DS

Arrays

Arrays: finite, ordered collection of elements stored in contiguous memory locations.

Properties:

1) Finite
2) Ordered
3) homogeneous
4) Static memory allocation

Operations

1) accessing
   a) sequential - $O(n)$
   b) random - $O(1)$

2) insertion
   a) best case - $\Omega(1)$
      (in end)
   b) worstcase - $O(n)$
      (at front)

3) deletion
   a) bestcase - $\Omega(1)$
      (in end)
   b) worst case - $O(n)$
      (at beginning)

4) Searching

a) unsorted - O(n)

b) sorted - O(log n)

Memory representation & addressing functions

format: base + size(value)

- describes how an array is stored in memory

I - Basic 1D array: (Little endian format, int size = 2)

arr [1, 2, 3, 4]

base address = 1000

size of element = 2

1	1000
2	1002
3	1004
4	1006

to generalize how this storing happens we write addressing functions

arr [0] → 1000

arr [1] → 1002

arr [2] → 1004

arr [3] → 1006

$$arr[i] = base + size(i)$$

note:

in C language:

```
int arr[5];
arr[6] = 10; // does not throw error
// starting add = 1000;
```

- in C array addressing happens only based on base address (not range)
- base address is applied in the addressing function and as long as the result of the function is a valid memory address, in this case

$$add[6] = 1000 + 2(6) \rightarrow 1012$$

- the value gets stored in 1012 because the compiler does not keep track of range and does not know that 1012 is not a part of the array.
- the actual problem arises when we try to access arr[6] because the memory location 1012 doesnot belong to the array, the compiler might rewrite another variable on that location
- hence it will give unpredictable results but not an error.
- but this is handled in Java using array index out of bounds expression.

II - 2D Array Storage

1) Row major order

2) Col major order

1) Row major order

arr $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ ⟶

1	1000
2	1002
3	1004
4	1006
5	1008
6	1010
7	1012
8	1014
9	1016

how addressing functions are written in general

$$\text{address} = \text{base address} + \text{size of element} \times (\text{number of Preceding elements})$$

number of Preceding elements = number of elements in Prev. rows + number of elements in Preceding colums of the i^{th} row

in this case:

- to find the location of a Particular element first you need to know how many elements are Preceding it.

- consider the example of finding location of element 5 (arr[1][1])
- 5 is in the second row (i.e i=1) which means there is a whole row of elements before it. this can be computed as

  = (i x number of columns)

  = (i x cols) // 1 x 3 = 3 elements in the row preceding it

- now compute the element before 5 in it's own row: (∵ j = 1) → it indicates that there is one element before it. = 1
- now, total number of elements before 5 (arr[1][1]) is (3+1 = 4) ⇒ (i x cols + j)
- Now just put it in the format

address = base address + size of element x (i x cols + j)

arr[1][1] = 1000 + 2 (1 x 3 + 1) → 1008 ✓

2) col major order

$$arr = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

(i → row, j → column)

=

1	1000
4	1002
7	1004
2	1006
5	1008
8	1010
3	1012
6	1014
9	1016

-> consider 5 arr[i][i] to get to it first we have to cover previous column elements

(row x j)

-> now cover the preceeding elements in the current col ⇒ i

-> total number of preceeding elements : (row x j + i)

-> addressing function: base address + size (row x j + i)

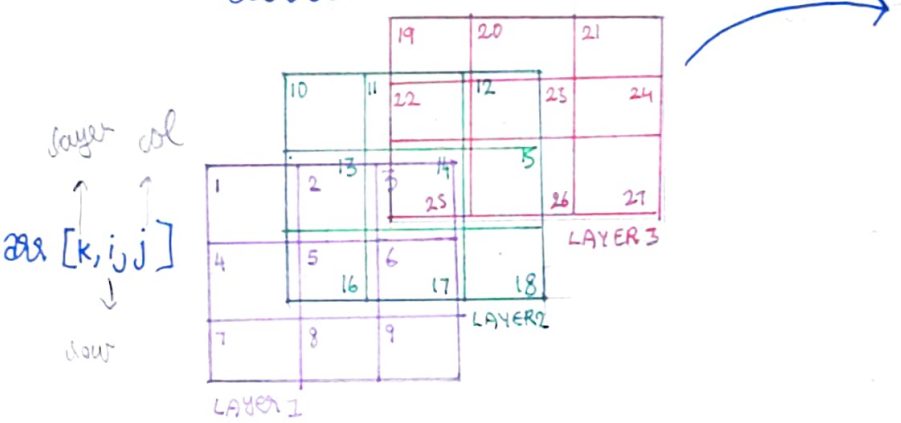
row major

1	1	1000
2	4	1002
3	7	1004
4	2	1006
5	5	1008
6	8	1010
7	3	1012
8	6	1014
9	9	1016
10	10	1018
11	13	1020
12	16	1022
13	11	1024
14	14	1026
15	17	1028
16	12	1030
17	15	1032
18	18	1034
19	19	1036
20	22	1038
21	25	1040
22	20	1042
23	23	1044
24	26	1046
25	21	1048
26	24	1050
27	27	1052

col major

III - 3d array memory representation:

3 layers of 2D Matrix



1) calculate number of elements in previous layer

2) calculate number of elements in previous row/cols in current layer

3) calculate number of elements preceeding in current row/col in current layer.

(i) Row major order:

→ consider element 14 → 222 [1,1,1]

1 x 3 x 3 = 9

→ number of elements in before the layer = k * rows * cols

→ number of elements in Previous rows = i x cols
current layer) 1 x 3 = 3

→ number of elements in Prev cols in ith row = j = 1

→ total number of Preceding elements = k * col * row + (i * col + j)

9 + 3 + 1 = 13

→ substituting in the addressing format

$$\text{address} = \underset{\text{add}}{\text{base}} + \text{size}\,(k * col * row + (i * cols + j))$$

1000 + 2(13) → 1026

→ similarly for column major order:

$$\text{address} = \underset{\text{add}}{\text{base}} + \text{size}\,(k * rows * cols + (j * rows + i))$$

special matrices

→ writing special addressing functions inorder to store special matrices effeciently.

Eg 1: Diagonal matrix:

$$arr = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

only storing 4 elements

1	1000
2	1002
3	1004
4	1006

arr [0] [0] = 1000
arr [1] [1] = 1002
arr [2] [2] = 1004
arr [3] [3] = 1006

Observe the pattern and write and addressing functions with conditions

condition:

when $i = j$

Address = base + size (i)

else

arr [i] [j] = 0

Eg: 2 band matrix

$$arr = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 \\ 0 & 0 & 9 & 10 \end{bmatrix}$$

storing only 10 elements

1	1000
2	1002
3	1004
4	1006
5	1008
6	1010
7	1012
8	1014
9	1016
10	1018

- condition: non zero when

$$\left.\begin{array}{l} i-j=0 \\ i-j=1 \\ i-j=-1 \end{array}\right\} \quad |i-j| \leq 1$$

- non zero elements : 10 which can be written as

[n = 4]

3n - 2 non zero elements

3(4) - 2 ⇒ 12 - 2 = 10

- calculating number of elements in previous rows when

i = 0 ⇒ 0

i = 1 ⇒ prev row has 2 elements

i = 2 ⇒ 2 + 3 = 5

i = 3 ⇒ 2 + 3 + 3 = 8

i = 0 1 2 3

0, 2, 5, 8

follows the pattern

(3i - 1) elements

- calculating number of elements preceeding in current row when

i = 0 ⇒ there are 2 elements in i^{th} row

i = 1 ⇒ 3

i = 2 ⇒ 3

i = 3 ⇒ 2

take example element

→ 4 // arr[1][1] → has 1 element preceding (1-1+1)

→ 5 // arr[1][2] → has 2 elements preceding (2-1+1)

→ 8 // arr[2][3] → has 2 elements preceding (3-2+1)

see these follow a pattern of $\boxed{j-i+1}$

- total number of preceding elements

$$3i-1 + j-i+1$$

$$\boxed{2i+j}$$

- addressing function:

arr[i][j] = base + size $(2i+j)$

when $|j-i| \leq 1$

else

arr[i][j] = 0

Eg: 3 super diagonal matrix

$$arr = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

1	1000
2	1002
3	1004

only 3 memory spaces need : i.e $(n-1)$ spaces

condition : only when $j = 1+i$

arr [0][1] → 1000

arr [1][2] → 1002

arr [2][3] → 1004

Observing the pattern

addressing function:

base + size (i)

add

```
if j = 1+i
   arr[i][j] = base + size(i)
else
   arr[i][j] = 0
```

Eg : 4 minor diagonal matrix

$$arr \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix} \Rightarrow$$

1	1000
2	1002
3	1004
4	1006

arr[0][3] → 1000

arr[1][2] → 1002

arr[2][1] → 1004

arr[3][0] → 1006

```
if j+1 = n-1
   arr[i][j] = base + size(i)
else: arr[i][j] = 0
```

Eg 5: upper triangular matrix:

$$arr = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

$i \le j$

1	1000
2	1002
3	1004
4	1006
5	1008
6	1010
7	1012
8	1014
9	1016
10	1018

add = base + size (no. of pre element)

no. of elements in pre rows + no. of elements in pre cols of same row

(i) when i =

0 → 0

1 → 4 = 4 → row - 0

2 → 4+3 = 7 → (row - 0) + (row - 1)

3 → 4+3+2 = 9 → (row - 0) + (row - 1) + (row - 2)

in general $\sum_{K=0}^{i-1} (row - K)$ where K = 0 to i-1

(ii) number of elements in the i^{th} row

take example elements and observe pattern

i = 0 → 4

i = 1 → 3

i = 2 → 2

i = 3 → 1

3 → arr[0][2] → has 2 elements before it → 2 - 0 = 2

6 → arr[1][2] → has 1 element before it → 2 - 1 = 1

pattern = j - i

no. of pre elements in total = $\sum_{K=0}^{i-1} (row - K) + (j - i)$

Addressing function:

when $i \geq j$

$$arr[i][j] = \text{base address} + \text{size}\left[\sum_{k=0}^{i-1}(row-k) + (j-i)\right]$$

else

$$arr[i][j] = 0$$

Eg6: lower triangular matrix

$$arr = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{bmatrix} \quad i \geq j$$

1	1000
2	1002
3	1004
4	1006
5	1008
6	1010
7	1012
8	1014
9	1016
10	1018

(i) no. of elements in pre rows

$i=0 \Rightarrow 0$
$i=1 \Rightarrow 1$
$i=2 \Rightarrow 1+2$
$i=3 \Rightarrow 1+2+3$

$\sum_{K=0}^{i} K$ (summation of numbers from 0 to i)

(ii) no. of elements in pre cols of current row

$i=0 \Rightarrow 1$
$i=2 \Rightarrow 2$
$i=3 \Rightarrow 3$
$i=4 \Rightarrow 4$

eg// 3 arr[1][1] → 1
// 6 arr[2][2] → 2

j

total no. of pre elements $\Rightarrow \left(\sum_{K=0}^{i} K + j\right)$

Addressing function:

when $j \geq i$

$$arr[i][j] = \text{base address} + size\left(\left(\sum_{k=0}^{i} K\right) + j\right)$$

else

$$arr[i][j] = 0$$

Eg 7: symmetric matrix

$arr =$

	0	1	2	3
0	1	2	3	4
1	2	5	6	7
2	3	6	8	9
3	4	7	9	10

1	1000
2	1002
3	1004
4	1006
5	1008
6	1010
7	1012
8	1014
9	1016
10	1018

only have to store 10 elements instead of 16; ~~and~~ ~~return~~ here 12 elements will be mapped 6 different addresses (i.e 2 elements same address) & main diagonal is stored once

[6+4=10]

elements with same mapping:

[0,1] & [1,0], [1,2] & [2,1]

[0,2] & [2,0], [1,3] & [3,1]

[0,3] & [3,0], [2,3] & [3,2]

hence only store this + main diagonal

[1,1] [2,2] [3,3] [4,4]

From this, the condition is $j-i \leq n-1$ & $j-i \geq 0$

$$0 \leq j-i \leq n-1$$

$$\downarrow \qquad i \leq j$$

(i) number of elements in previous row [similar to upper triangular matrix]

$i=0 \quad 0$

$i=1 \Rightarrow 4$

$i=2 \Rightarrow 4+3$

$i=3 \Rightarrow 4+3+2$

$$\left(\sum_{k=0}^{i-1} row - k\right)$$

(ii) number of elements in current row

$i=0 \quad 4$

$i=1 \Rightarrow 3$

$i=2 \quad 2$

$i=3 \quad 1$

$\because$ to upper Δ - $j-i$

total number of pre elements

$$\sum_{k=0}^{i-1}(row-k) + (j-i)$$

(iii) addressing function:

when $0 \leq j-i \leq n-1$ & $i \leq j$

$$arr[i][j] = base + size * \left(\sum_{i=0}^{k}(row-k) + (j-i)\right)$$

else

$$arr[i][j] = arr[j][i]$$

Eg: 8 a Exchange matrix (similar to minor diagonal)

$$arr = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

```
when  j+i=n-1
      arr[i][j]=1
else
      arr[i][j]=0
```

Eg: 9 upper shift matrix (similar to superdiagonal)

$$arr = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

```
when  i=j-1
      arr[i][j]=1
else
      arr[i][j]=0
```

Eg: 10 lower shift matrix (similar to sub diagonal)

$$arr = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```
when  i=j+1
      arr[i][j]=1
else
      arr[i][j]=0
```

Eg:9 Toeplitz matrix elements on the diagonal are same.

arr =

i \ j	0	1	2	3
0	4	3	2	1
1	5	4	3	2
2	6	5	4	3
3	7	6	5	4

4	1000
3	1002
2	1004
1	1006
5	1008
6	1010
7	1012

- total $2n-1$ elements have to be saved instead of n^2

- only elements on the first row/column are saved
  (ie. $i=0$ or $j=0$)

(i) number of pre elements in previous rows

i	n
0	0
1	4
2	4+1 → 4+1
3	4+1+1 → 4+2

} $n+(i-1)$

(ii) number of pre elements in current row

i	n
0	4
1	1
2	1
3	1

n

[0,0] → 0
[0,1] → 1
[0,2] → 2
[0,3] → 3
[1,0] → 0
[2,0] → 0
[3,0] → 0

} j

total number of pre elements : $n+(i-1)+j$

elements are stored at

base + size (n + (i-1) + j) location

- retrieving elements from the upper triangular matrix after storage

  i.e when $i \leq j$

  eg → [1,1] $\xrightarrow{(4)}$ 1000
  [2,3] $\xrightarrow{(3)}$ 1002
  } base + size (j - i)

- retrieving elements from the lower half after storing i.e. $i > j$

  eg → [2,0] $\xrightarrow{[6]}$ 1010
  → [3,1] $\xrightarrow{[6]}$ 1010
  } base + size (n + (i - j - 1))

addressing function:

when i == 0 or j == 0

arr[i][j] = base + size (n + (i - 1) + j)

when $i \leq j$

arr [i] [j] = base + size (j - i)

when $j > i$

arr [i] [j] = base + size (n + (i - j - 1))

Eg: 10 sort an array with 0's and 1's

sample input → [1,0,0,1,1,0,1,0]

sample output → [0,0,0,0,1,1,1,1]

for i=0 to n count the number of zeros // 4

```
if i<count
    arr[i] = 0
else
    arr[i] = 1
```

Eg: 11 sort array of 0s, 1s, 2s

input → [1,0,2,1,2,0]

output → [0,0,1,1,2,2]

for i = 0 to n count the number of 0's
count2 ⇒ counts number of 1's

```
if i<count
    arr[i] = 0
if i≥count && i< count2
    arr[i] = 1
else
    arr[i] = 2
```