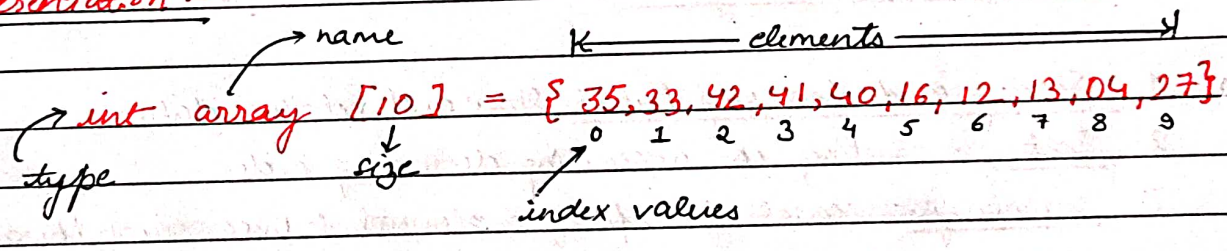
Arrays

Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. (sequential and uninterrupted manner) 4
Can be randomly accessed using index numbers.

Properties :-

1. Homogeneous elements: All elements share the same data type and size (one by one, side-side)
2. Contiguous memory: elements are stored adjacently in memory
3. Fixed Size: The array has pre-determined and fixed size, can't be changed
4. Random Access: Elements can be accessed using indices.
5. Uniform size: Each element occupies the same amount of memory

Representation :-



Operation :-

1. Traversal :- print the elements of the array.

```
for (int i=0; i< array.length ; i++){
    System.out.println(array[i]); }
```

2. Insertion :- Add element at a particular index.

```
for (int i= array.length-1 ; i>index ; i--){
    array[i] = array[i-1] ; }
array[index]=value ;
```

3. Deletion :- Delete element from particular index

```
for (int i = index; i < array.length - 1; i++){
    array[i] = array[i+1]; }
array = Arrays.copyOf(array, array.length - 1);
```

4. Search :- It search an element using the given index or by the value.

```
for (int i = 0; i < array.length; i++){
    if (array[i] == key){
        return i; }}
```

5. Update :- array element gets updated.

```
array[index] = newValue;
```

Application of Arrays :-

1. Storing and Accessing Data :- Store and retrieve data
2. Sorting :- sorting in accending, decending order
3. Searching :- search for specific elements :- linear seach / binary search
4. Matrice :- represent matrice in mathematical computations such as matrix multiplic
5. Stack and Queue :- used as underlying data structure for queue and stack.
6. Graph :- Represent Graphs.
7. Dynamic Programming :- To store intermidiate results.

Real-Time applications :-

1. Signal Processing :-
   Arrays handle time-series samples in speech recognition and radar system.
2. Multimedia Applications :-
   Array store pixel / audio samples, e.g., RGB values in image processing
3. Data Mining :-
   large datasets for efficient real-time processing.

4. Robotics :-
3D-object positioning, motion planning, object recognition.
5. Monitoring and Control Systems :-
arrays store sensor data, enabling real-time decision making in automation
6. Financial Analysis :-
store historical stock prices, supporting efficient real-time analysis.
7. Scientific Computing :-
Arrays store numerical data for real-time analysis in experiments.

Array Application in Java :-

1. Store collection of same type.
2. Implement matrice and table using 2D arrays.
3. Perform sorting and searching operations.
4. Serve as underlying data structures for stack, queue and heaps.
5. Commonly used for image processing for storing pixel values.

backbone

Array Advantages :-

1. Efficient O(1) to access elements.
2. Fast data retrieval due to contiguous memory allocation.
3. Memory efficiency with known size at compile time.
4. Easy implementation

Disadvantages :-

1. Fixed size
2. Inefficient insertion and deletion.
3. Wasted space for unpopulated elements.
4. Limited support.
5. Lack of flexibility like other data structure.

Advantages of other structure over arrays :-

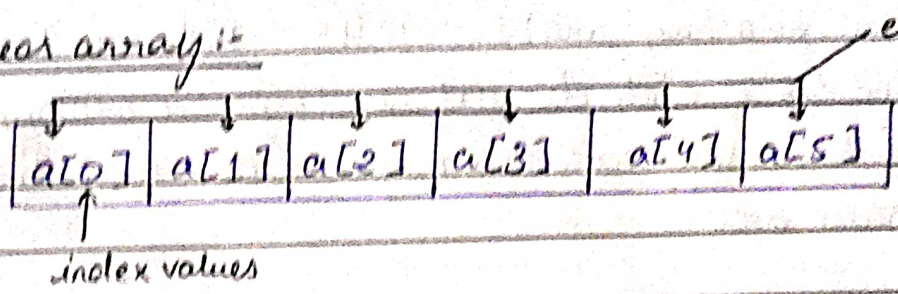
1. The structure can store different types of data whereas an array can only store similar data types.
2. Structure doesnot have limited size like an array.
3. Structure elements may or maynot be stored in continuous memory location but array elements are stored in contiguous locations.

4.1

Calculation of Address of element of 1D - 2D and 3D using row major and column major order :-

B = Base Address or start address
W = weight (storage size of one element stored in array)
M = Row (Total number of rows)
N = Columns (total number of columns)
P = Width (Total number of cells depth-wise)
x = Lower Bound of Block (first subscripts)
y = Lower bound of row
z = Lower bound of column
I = Row in 3D to find / Row subset in 1D-2D whose address to be found
J = Row in 3D to find / Column subset in 1D-2D whose address to be found
K = column 3D to find address
Formula of M (Row) ⇒ Upper bound (row) - Lower bound (row) + 1
Formula of N (column) ⇒ Upper bound (col) - Lower bound (col) + 1

1-D or linear array :-



Address of A[I] = B + W * (I - LB)

Example → Given base address of an array A[1300 .... 1900] as 1020 and size of each element is 2 byte in the memory find the address of A[1700]

B = 1020
LB = 1300
W = 2 byte
I = 1700

A[1700] = 1020 + 2 * (1700 - 1300)
= 1020 + 2 * 400
= 1020 + 800
A[1700] = 1820

2-D array :-

	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]

rows ↓ columns →

Row Major Order →

Address of A[I][J] = B + W * ((I - LR) * N + (J - LC))

Column Major Order →

Address of A[I][J] = B + W * ((J - LC) * M + (I - LR))

Example → Given an array, arr[1 ..... 10][1 .... 15] with base value 100 and size of each element is 1 Byte in memory. Find address of arr[8][6] with row major order and then column major order.

B = 100
W = 1 Byte
I = 8
J = 6
LR = 1
LC = 1

Number of column in given matrix (N) = Upper bound - lower Bound + 1
= 15 - 1 + 1
= 15

Number of rows in given matrix (M) = Upper bound - lower Bound + 1
= 10 - 1 + 1
= 10

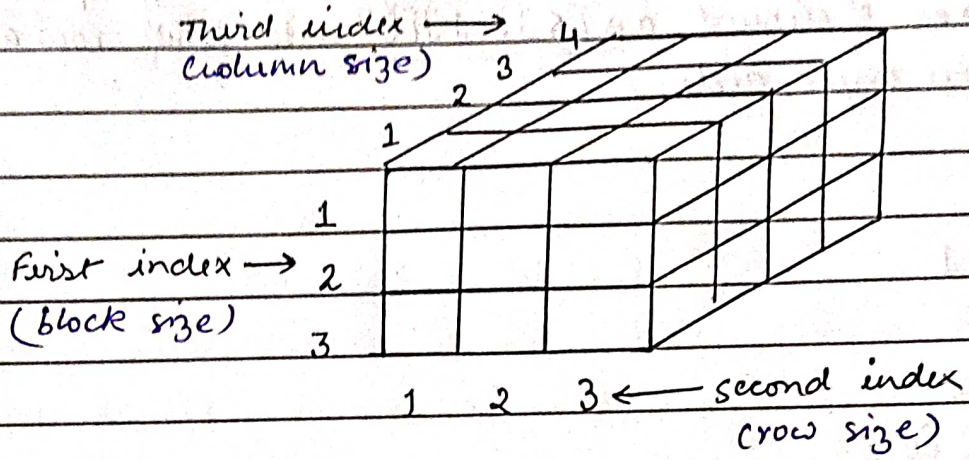
Row major order ⇒

Address of A[I][J] = B + W * ((I - LR) * N + (J - LC))
Address of A[8][6] = 100 + 1 * ((8 - 1) * 15 + (6 - 1))
= 100 + 1 * ((7) * 15 + (5))
= 100 + 1 * (110)
= 210

Column Major Order ⇒

Address of A[I][J] = B + W * ((J - LC) * M + (I - LR))
Address of A[8][6] = 100 + 1 * ((6 - 1) * 10 + (8 - 1))
= 100 + 1 * ((5) * 10 + (7))
= 100 + 1 * (57)
= 157

3-D or collection of 2D arrays :-



3-D array with 36 elements

declare {
int arr [width][height][depth] ;
int arr [block index][row index][column index] ;
int arr [3][3][4] ;

Row Major Order :-

Address of A[I][J][K] = B+w * (M*N*(I-x)+N(J-y)+(K-z))

Column Major Order :-

Address of A[I][J][K] = B+w*(M*N(I-x]+M*(K-z)+(J-y))

example of 3D next page →

Example → Given an array, arr[1:9, -4:1, 5:10] (row, column) with base value of 400 and each element size is 2 byte in memory find the address of element arr[5][-1][8] with either row and column major orders?

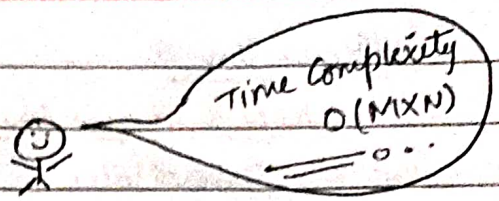
I = 5
J = -1
k = 8
B = 400
W = 2 Byte
X = 1
y = -4
z = 5
M (row) = 1-(-4)+1 = 6
N (column) = 10-5+1 = 6

Row Major →

Address of [I][J][K] = B+W(M*N(I-X)+N*(J-Y)+(K-Z))
Address of [5][-1][8] = 400+2*((6*6*(5-1))+6*((-1+4)))+[8-5]
= 400+2*(6*6*4)+(6*3)+3
= 400+2*(144)+(21)
= 400+2*(165)
= 730

Column Major →
Address of [I][J][K] = B+W(M*N(I-X)+M*(K-Z)+(J-Y))
Address of [5][-1][8] = 400+2*(6*6(5-1)+6*(8-5)+(-1-(-4)))
= 400+2*(144+18+3)
= 400+2*(165)
= 730

Sparse Matrix :



Sparse Matrix are those matrices which have mostly zero elements. general means greater number of zeros than non zeros.

Why sparse Matrix is required :-

Storage :- Sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only non-zero elements.

Computing Time : It saves searching time as it only traverses non-zero elements.

Array Representation of Sparse Matrice :-

ROW	COL	VALUE

→ value at index(row, col)

Row Index

Column Index

	0	1	2	3 ← Columns
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	0	0	0
4	1	0	0	0

SPARSE MATRIC E

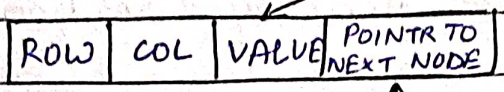
Rows

Table Structure :-

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

Linked list implementation of stack :-

Node Structure



non-zero element located at index (row, column)

stores address of next node

	0	1	2	3
0	0	0	1	0
1	3	0	0	0
2	0	4	5	0
3	0	6	0	0

SPARSE MATRIX