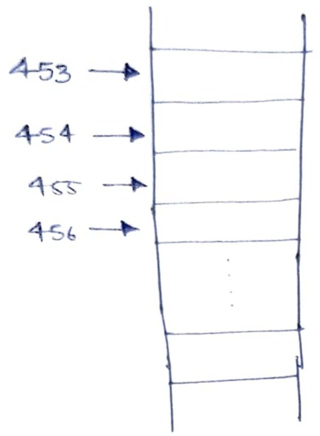
ARRAY.

↳ If we want to store a group of data together in one place, then an array is the Data structure we should be looking for. In this Data structure, all elements are stored in contiguous locations of the memory.



AN ARRAY OF DATA.

Definition.

An array is a finite, ordered and collection of homogeneous data elements.

→ Why finite → contains only limited number of elements

→ Why ordered → stored one by one in contiguous locations of the computer memory in a linear ordered fashions.

→ Why homogeneous → elements of an array are of same data types.

Terminology of an Array.

(1). Size → No. of elements in an array.
↳ aka LENGTH or DIMENSION.

(2). Type → Type of an array represents the kind of data type it is meant for

(3). Base → Base of an array is the address of the memory location where the first element of the array is located.

(4). Index → All elements in an array can be referenced by a subscript like A_i or $A[i]$

→ an Index is always an Integer value.

→ As each array element is identified by a subscript or index, an array element is also termed subscripted or indexed variable.

(5). Range of INDICES →
Indices of array elements may change from lower bound (L) to an upper bound (U), and these bounds are called the Boundaries of an array.

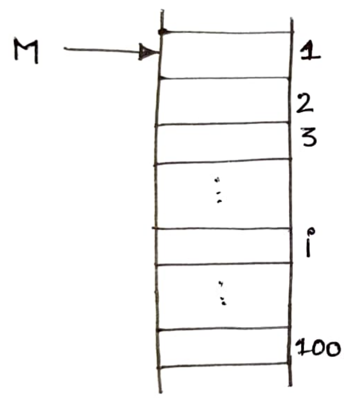
ONE-DIMENSIONAL ARRAY.

→ if only one subscript / index is required to reference all the elements in an array, then the array is termed ONE DIMENSIONAL ARRAY OR SIMPLY An array. Accessing its elements involves a single subscript that can either represent a row or column index.

Memory Allocation for an Array.

Let the memory location where the first element is to stored be M.
If each element requires one word, the ~~th~~ location for any element say A[i] in the array can be obtained as

$$\text{Address}(A[i]) = M + (i-1)$$



Physical Representation of a One Dimensional Array.

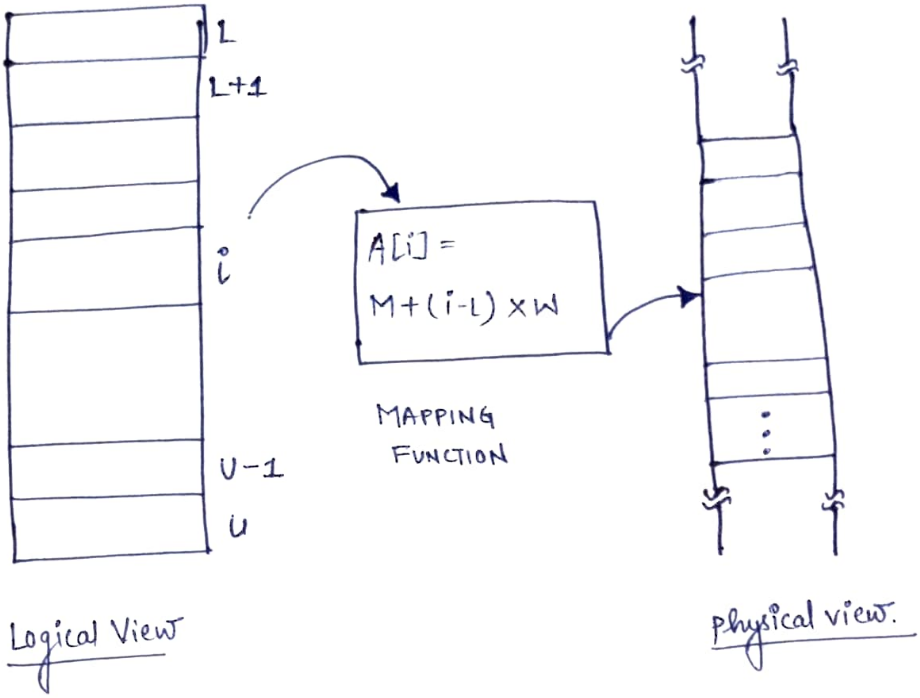
Likewise, in general, an array can be written as A[L...U], where L and U denote the lower and upper bounds for the index.
If the array is stored starting from the memory location M, and for each element it requires W number of words for A[i] will be

$$\text{Address}(A[i]) = M + (i-L) \times W$$

W = aka → storage size of one element store in an array (in bytes).

This is known as Indexing formula → which is used to map the logical presentation of an array to physical presentation.

By knowing the starting address of an array M, the location of the ith element can be calculated instead of moving towards i from M.



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

Soln ⇒ Base Address = 1020

Lower Limit (L) = 1300

W = 2 Bytes

Subset of element whose address to be found I = 1700.

Formula used → $A[i] = M + (i - L) * W$

Address of $A[1700] = 1020 + (1700 - 1300) * 2$

$= 1020 + 2 * (400)$

$= 1820$

Address of A[1700] = 1820.

Calculate Address of any element in the 2-D array

	Columns 0	1	2
Rows 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]

To find the address of any element in a 2-Dimensional Array there are two ways:—

1. Row Major Order → elements of an array are stored in Row-wise fashion.
2. Column Major Order
   ↳ elements of an array are stored in a column major fashion means moving across the column and then to the next column then it's in column major order.

Row Major Order

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

i = Row Subset of an element whose address to be found.

J = Column subset of an element whose address to be found.

B = Base Address

W = Storage Size of one element store in an array (in bytes)

LR = Lower Limit of row/start row index of the matrix

LC = " " " column/start column " " " "

N = No. of column given in the matrix.

Example → Given an array $arr[1 \ldots 10][1 \ldots 15]$ (Row, Column) with base value of 100 and size of each element is 1 Byte in memory. Find the address of $arr[8][6]$ with the help of row-major order.

→ Base address = 100
→ W = 1 Bytes.
→ Row Subset I = 8
→ Column " J = 6
→ LR = 1
→ LC = 1
→ Number of column given in the matrix N = UpperBound − LowerBound + 1.

$$= 15 - 1 + 1$$
$$= 15$$

Formula :→

$$\text{Address of } A[i][j] = B + W * (i - LR) * N + (J - LC)$$
$$= 100 + 1 \times (8 - 1) * 15 + (6 - 1)$$
$$= 100 + 105 + 5$$

$$\boxed{\text{Address of } A[i][j] = 210}$$

2-D column major order:

$$\boxed{\text{Address of } A[I][J] = B + W * ((J - LC) * M + (I - LR))}$$

→ B = Base address = 100
→ I = Row Subset of an element whose address to be found. = 8
→ J = Column Subset of an element whose address to be found. = 6
→ B = Base Address = 100
→ W = storage size of one element store in any array = 1 Bytes
→ LR = Lower limit of row / start row index of matrix = 1
→ LC = Lower limit of column / start column " " = 1
→ M = Number of rows given in the matrix. = U.B − L.B + 1 = 10 − 1 + 1 = 10.

Address of A[I][J] = 100 + 1 * (6 − 1) * 10 + (8 − 1)
= 100 + 1 × 5 * 10 + 7
= 157

Address of A[I][J] = 157.

Operations on Arrays.

(1) Traversing

Algorithm TraverseArray

Input: An array A with element

output: According to Process( )

Data structures: Array A[L.. U] // upper and lower bounds of array index.

Steps :-

1. i=L // start from the first location L
2. while i ≤ U do
3. Process (A[i])
4. i = i+1 // More to the next location.
5. EndWhile
6. Stop

Note :- here Process( ) is a procedure which when called for an element can perform an action. for ex → display the element on the screen, determine whether A[i] is empty or not etc.

Process( ) can also be used to manipulate some special operations such as count the special element of interest (for ex, negative numbers in an integer array), update each element of the array.

② Sorting an Array.

This operation, if performed on an array, will sort it in a specified order (ascending / descending).

following algorithm is used to store the elements of an integer array in ascending order.

Input :→ An array with integer data.

Output :→ An array with sorted elements in an order according to Order().

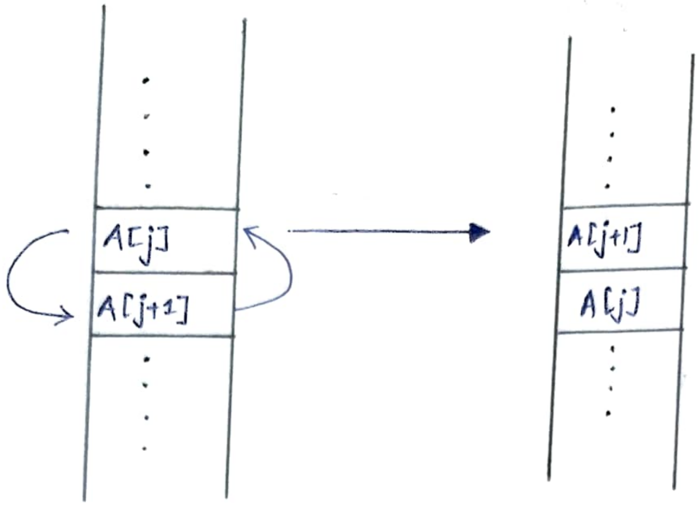
Data Structure :→ An integer array A[L...U]

// L and U are the lower and upper bounds of array index.

Steps

1. i=U
2. While i ≥ L do
3. j= L // start comparing from first
4. While j<i do
5. If Order (A[j], A[j+1]) = FALSE // If A[j] and A[j+1] are not in order
6. swap (A[j], A[j+1]) // Interchange the elements.
7. EndIf
8. j=j+1
9. End While
10. i=i-1
11. End While
12. Stop.

Here, order(...) is a procedure to test whether two elements are in order and Swap(...) is a procedure to interchange the elements b/w two consecutive locations.



Swapping of two elements in an array

3. Searching

This operation is applied to search an element of interest in an array.

A simplified version of the algo. is as follows :-

Algorithm SearchArray

Input : KEY is the element to be searched.

Output : Index of Key in A or a message on failure.

Data Structure :- An array A[L..U] // L and U are the lower and upper bounds of array index.

Steps :-

1. i=L , found=0, location = 0 // found = 0 indicates search is not finished and unsuccessful.

2. While (i≤U) and (found=0) do // continue if all or anyone condition does not satisfy.

3. If compare (A[i], ~~key~~ KEY) = TRUE then // If key is found

4. found = 1

5. location = i

6. Else

7. i = i+1

8. EndIf

9. End While

10. If found=0 then

11. print " Search is unsuccessful : KEY is not in the array"

12. ELSE

13. Print " Search is successful : KEY is in the array at location", location

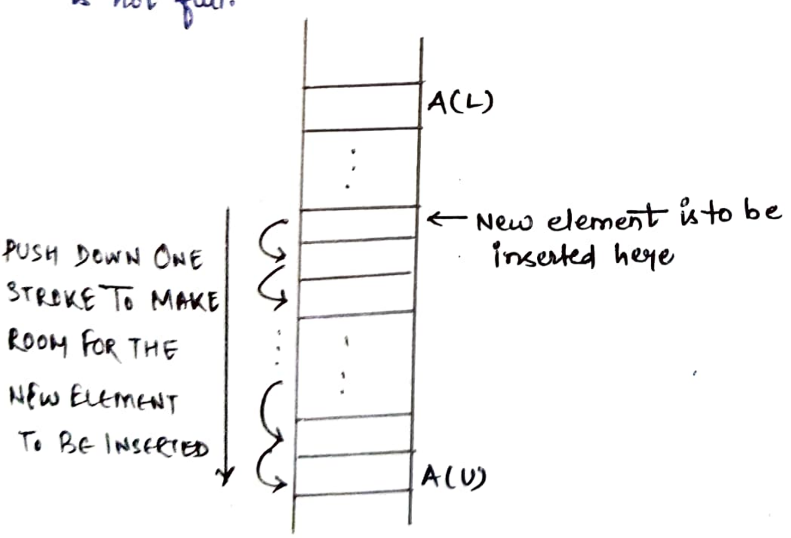
14. EndIf

15. Return (location)

16. Stop.

4. Insertion.

This operation is used to insert an element into an array provided that the array is not full.



ALGORITHM INSERT ARRAY

Input : KEY is the item, LOCATION is the index of the element where it is to be inserted.

Output: Array enriched with KEY.

Data Structure: An array A[L...U] // L and U are the lower and upper bounds of array index.

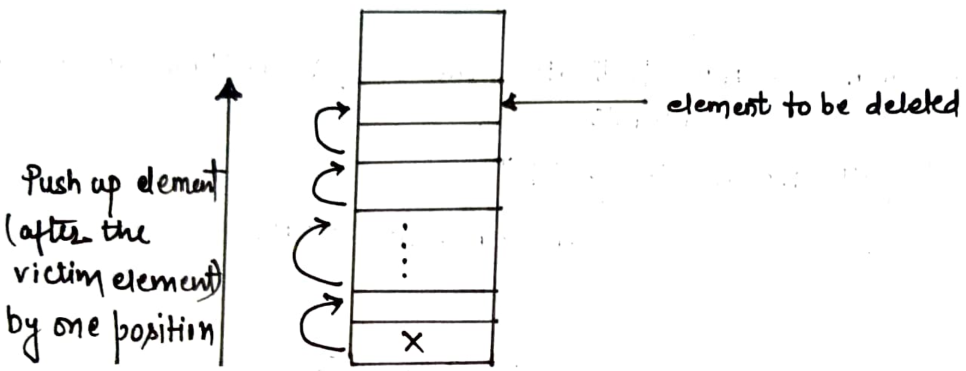
Steps:-

1. If A[U] ≠ NULL then // NULL indicates, the room is available for a new entrant.
2. PRINT "Array is full : No insertion possible"
3. EXIT // END OF EXECUTION.
4. ELSE
5. i = U // Start pushing from end.
6. While i > LOCATION do
7. A[i] = A[i-1]
8. i = i-1
9. END WHILE
10. A[LOCATION] = KEY // put the element at the desired location.
11. ENDIF
12. STOP

5. Deletion.

This operation is used to delete a particular element from an array. The element will be deleted by overwriting it with its subsequent element and this subsequent element then is also to be deleted.

In other words, push the tail one stroke up.



Deletion of an element from an array

ALGORITHM DeleteArray

Input: KEY the element to be deleted.

output: Slimed array without KEY

Datastructure: An array A[L...U]

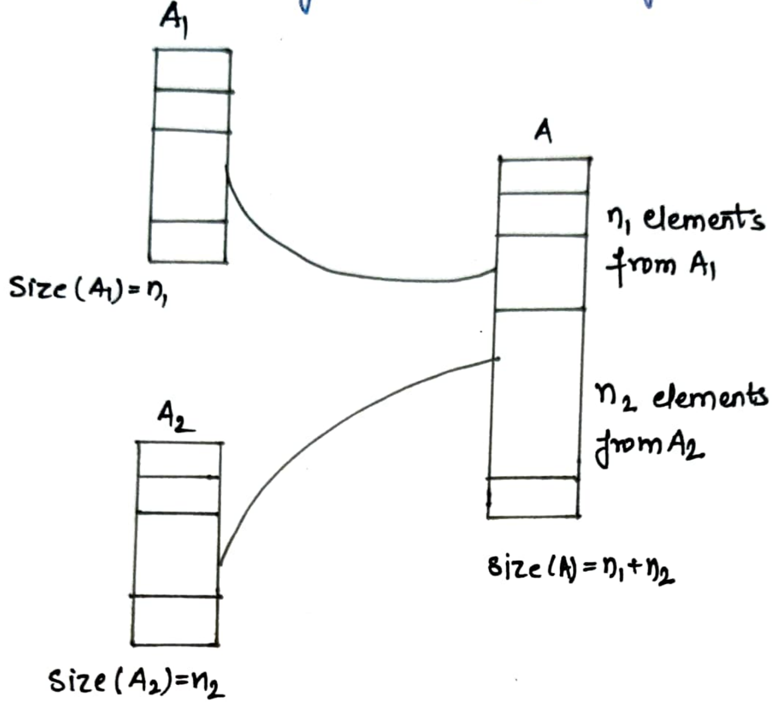
steps:

1. i= SearchArray (A, KEY) → perform search operation.
2. If (i=0) then
3. PRINT "KEY is not found : No deletion" → Exit Program
4. EXIT
5. ELSE
6. while i<U do
7. A[i]=A[i+1] → Replace element by its successor
8. i=i+1
9. EndWhile
10. ENDIF
11. A[U]= NULL → // The bottom-most element is made empty
12. U=U-1 → update the upper bound now;
13. Stop

NOTE → It is a general practice that no intermediate location will be made empty, that is, an array should be packed and empty locations are at the tail of an array.

6. Merging

Merging is an important operation when we need to compact the elements from two different arrays into a single array.



ALGORITHM MERGE

Input:- Two arrays $A_1[L_1 \ldots U_1]$, $A_2[L_2 \ldots U_2]$

output:- Resultant array $A[L \ldots U]$, where $L = L_1$ and $U = U_1 + (U_2 - L_2 + 1)$ when A_2 is appended after A_1

Data structures: Array structure.

Steps:→

1. $i_1 = L_1$, $i_2 = L_2$ ⟶ // Initialization of control variables
2. $L = L_1$, $U = U_1 + U_2 - L_2 + 1$ → // Initialization of lower and upper bound.
3. $i = L$
4. Allocate Memory (size $(U - L + 1)$) — // Allocate memory for the array A
5. While $i_1 \leq U_1$ do → // To copy array A_1 into the first part of A
6. $A[i] = A_1[i_1]$
7. $i = i + 1$, $i_1 = i_1 + 1$
8. EndWhile
9. While $i_2 \leq U_2$ do → // To copy array A_2 into the last part of A
10. $A[i] = A_2[i_2]$
11. $i = i + 1$, $i_2 = i_2 + 1$
12. Endwhile
13. Stop.

Three-Dimensional Array.

→ A three dimensional array can be compared with a book whereas two dimensional and one dimensional arrays can be compared with a page and a line, respectively.

→ Here, the Three major dimension can be termed row, column and page.

No. of rows = x (number of element in a column).

No. of column = y (no. of elements in a row).

no. of pages = z.

→ Storing a 3-D array means, storing the pages one by one. Again storing a page is the same as storing a 2-D array. Thus, if the element in a page are stored in row-major order then we term that 3-D array also in row-major order. The following formula is taking into consideration for row-major order of a 3-D array :→

Address (a_{ijk}) = No. of elements in the first $(K-1)$ pages
+
No. of elements in the K^{th} page upto $(i-1)$ rows
+
No. of elements in the K^{th} page, in the i^{th} row up to the j^{th} column

$$= xy(K-1) + (i-1)y + j$$

Instead of index starting from 1 for all indicies, if we assume that i changes between l_x and u_x, j changes b/w l_y and u_y and k changes b/w l_z and u_z, so that

$$x = u_x - l_x + 1$$
$$y = u_y - l_y + 1$$
$$z = u_z - l_z + 1$$

Also assuming the word size of each element to be w instead of 1 then the above indexing formula for a 3-D array can be restated in general form as

$$\text{Address}(a_{ijk}) = M + [xy(k - l_z) + (i - l_x)y + (j - l_y)] \times w$$

where M denotes the base address of the array.