UNIT-1

ONE-DIMENSIONAL ARRAY

Definition: 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.

Memory allocation for an array:

Suppose, an array A[100] is to be stored in a memory. Let the memory location where the first element is to be stored be M. If each element requires one word, then the location for any element say A[i] in the array is:

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

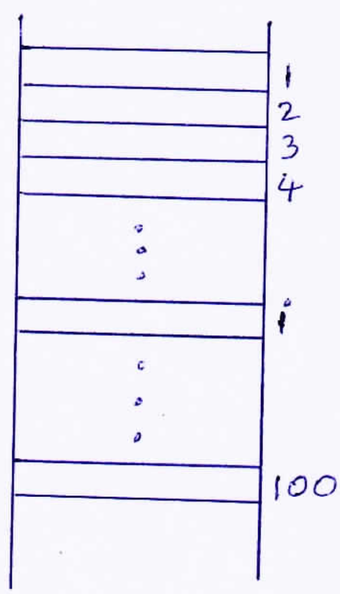

fig: Physical representation of a one-dimensional array

An array can be written as A[L ... U], where L and U are 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, then address for A[i] will be:

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

The above formula is known as indexing formula. It is used to map the logical presentation of an array to physical presentation.

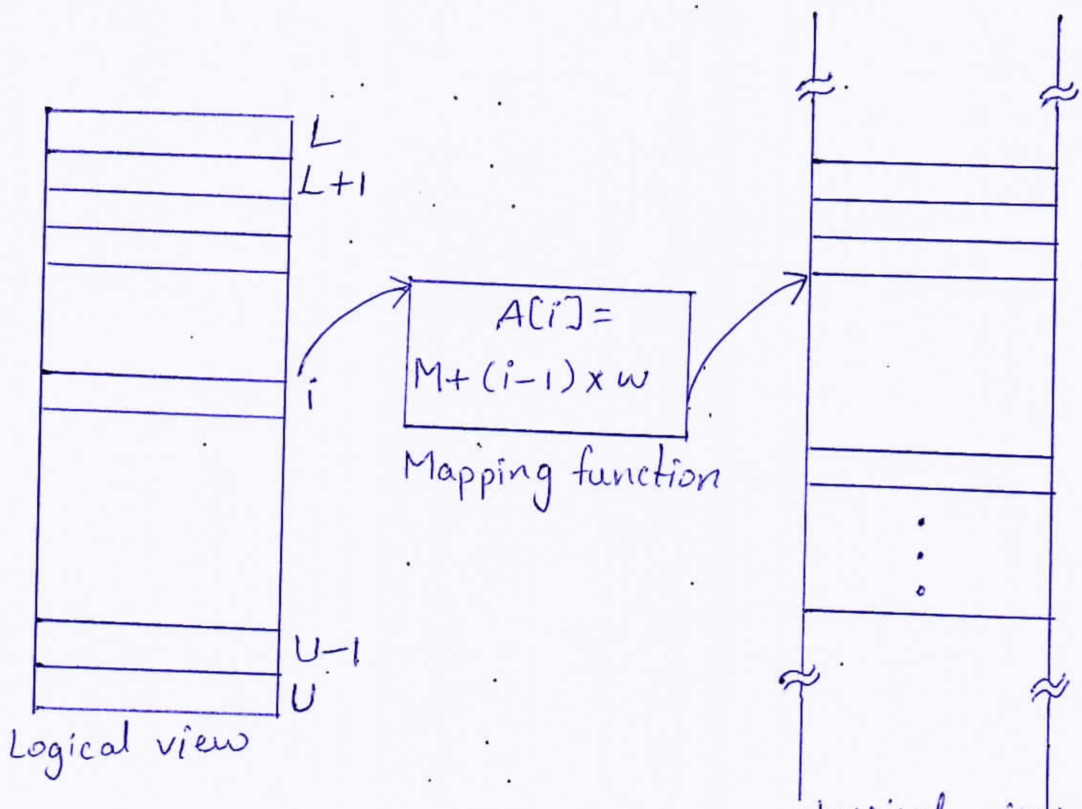

Fig: Address mapping between logical & physical views of an array.

Operations on Arrays:-

① **Traversing:** This operation is used to visit all elements in an array.

Algorithm TraverseArray.

Input : An array A with elements

Output : According to Process().

Data structures : Array A[L...U]

Steps :

1. i = L
2. while i ≤ U do
3. Process (A[i])
4. i = i+1
5. EndWhile
6. Stop

② **Sorting:** This operation is used to sort the elements of an array in a specified order (ascending / descending).

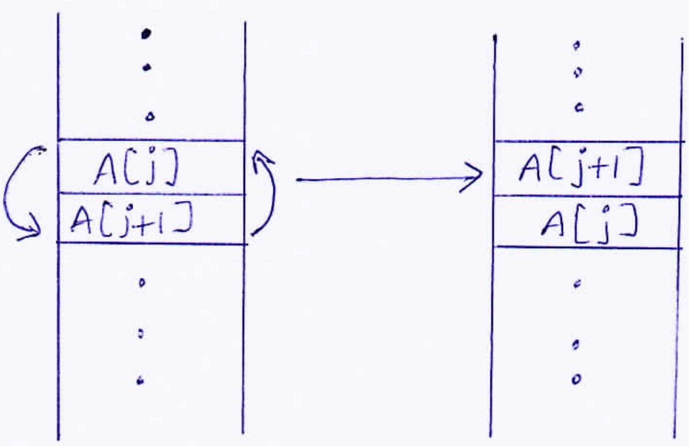

fig: Swapping of two elements in an array

Algorithm Sort Array

Input: An array with integer data.

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

Data Structures: An integer array A[L...U].

Steps:

```
1.  i = U
2.  While i ≥ L do
3.      j = L
4.      While j < i do
5.          If Order (A[j], A[j+1]) = FALSE
6.              Swap ( A[j], A[j+1])
7.          EndIf
8.          j = j+1
9.      EndWhile
10.     i = i+1
11. EndWhile
12. Stop
```

(3) **Searching:** This operation is used to search an element of interest in an array.

Algorithm SearchArray

Input: KEY is the element to be searched.

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

Data Structures: An array A[L...U].

Steps:

1. i = L , found = 0, location = 0
2. While (i ≤ U) and (found = 0) do
3. If Compare (A[i], KEY) = TRUE then
4. found = 1
5. location = i
6. Else
7. i = i + 1
8. EndIf
9. EndWhile
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.

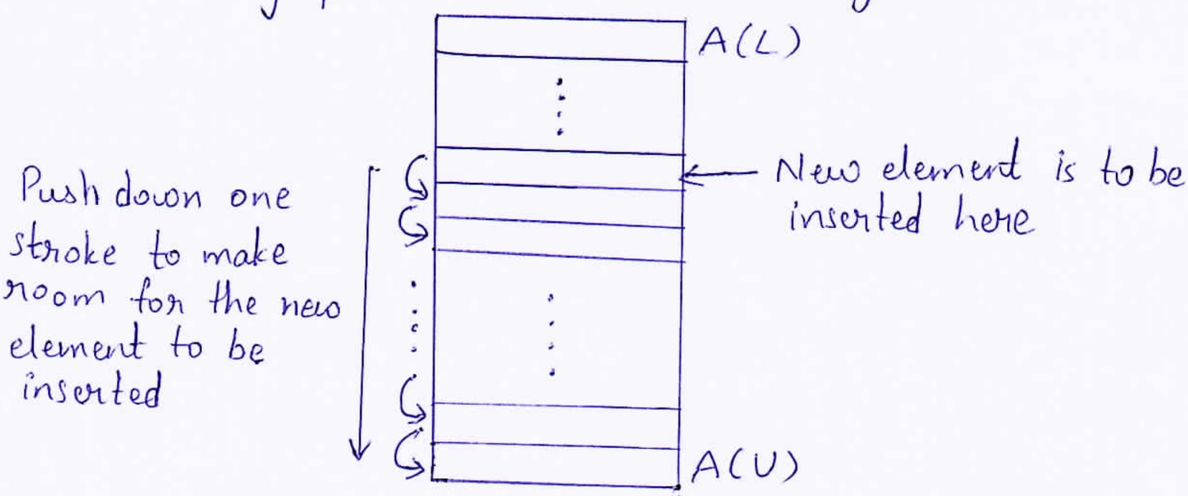

fig: Insertion of an element to an array

Algorithm InsertArray

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

Output: Array enriched with KEY.

Data structures: An array A[L ... U].

Steps:

```
1.  If A[U] ≠ NULL then
2.      Print "Array is full : No insertion possible".
3.      Exit
4.  Else
5.      i = U
6.      While i > LOCATION do
7.          A[i] = A[i-1]
8.          i = i-1
9.      EndWhile
10.     A[LOCATION] = KEY.
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 & then this subsequent element is also to be deleted.

⇒ In other words, push the tail one stroke up.

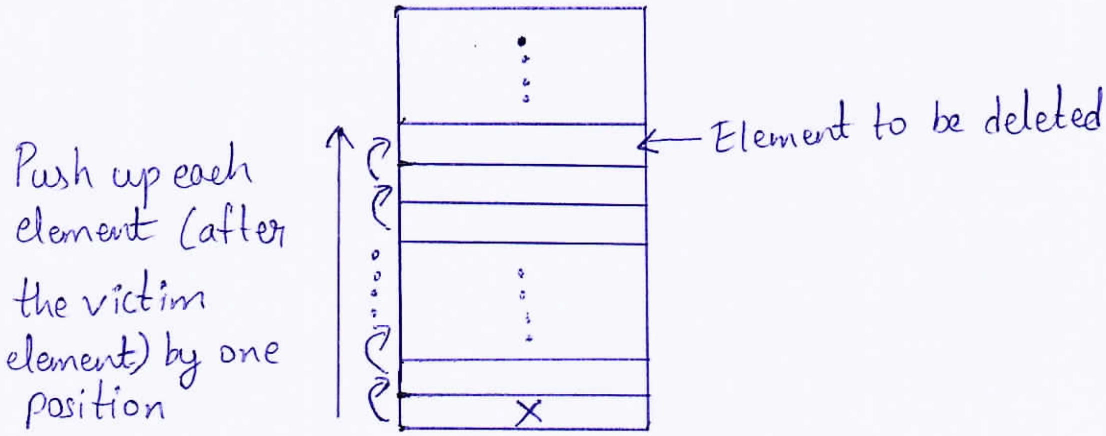

fig: Deletion of an element from an array.

Algorithm DeleteArray

Input: KEY the element to be deleted

Output: Slimmed array without KEY.

Data Structures: An array A[L...U].

Steps:

```
1.  i = SearchArray(A, KEY)
2.  If (i = 0) then
3.       Print "KEY is not found : No deletion"
4.       Exit
5.  Else
6.       While i < U do
7.            A[i] = A[i+1]
8.            i = i+1
9.       EndWhile
10. EndIf
11. A[U] = NULL
12. U = U-1
13. Stop
```

(6) **Merging:** This operation is used to compact the elements from two different arrays into a single array.

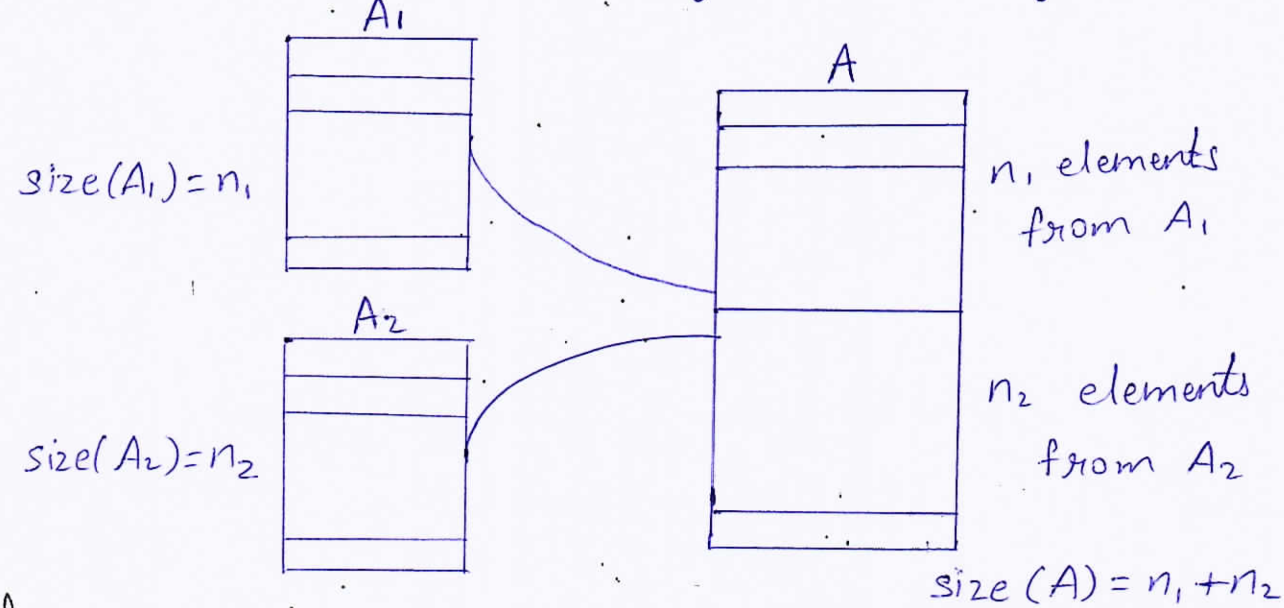

fig: Merging of A_1 and A_2 to A.

Algorithm Merge

Input: Two arrays $A_1[L_1 \cdots U_1]$, $A_2[L_2 \cdots U_2]$.

Output: Resultant array $A[L \cdots U]$, 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. $i_1 = L_1$, $i_2 = L_2$
2. $L = L_1$, $U = U_1 + U_2 - L_2 + 1$
3. $i = L$
4. AllocateMemory (Size (U-L+1))
5. While $i_1 \leq U_1$ do
6. $\quad A[i] = A_1[i_1]$
7. $\quad i = i+1$, $i_1 = i_1 + 1$
8. EndWhile

9. While $i_2 \le U_2$ do

10. $A[i] = A_2[i_2]$

11. $i = i+1$, $i_2 = i_2+1$

12. EndWhile

13. Stop.

Application of Arrays

The following is an example to store records of all students in a class. The record structure is given by:

STUDENTS

ROLL. NO.	MARK 1	MARK 2	MARK 3	TOTAL	GRADE
(Alphanumeric)	(Numeric)	(Numeric)	(Numeric)	(Numeric)	(Character)

If the sequential storage of records is not an objection, then we can store the records by maintaining 6 arrays whose size is specified by the total number of students in the class.

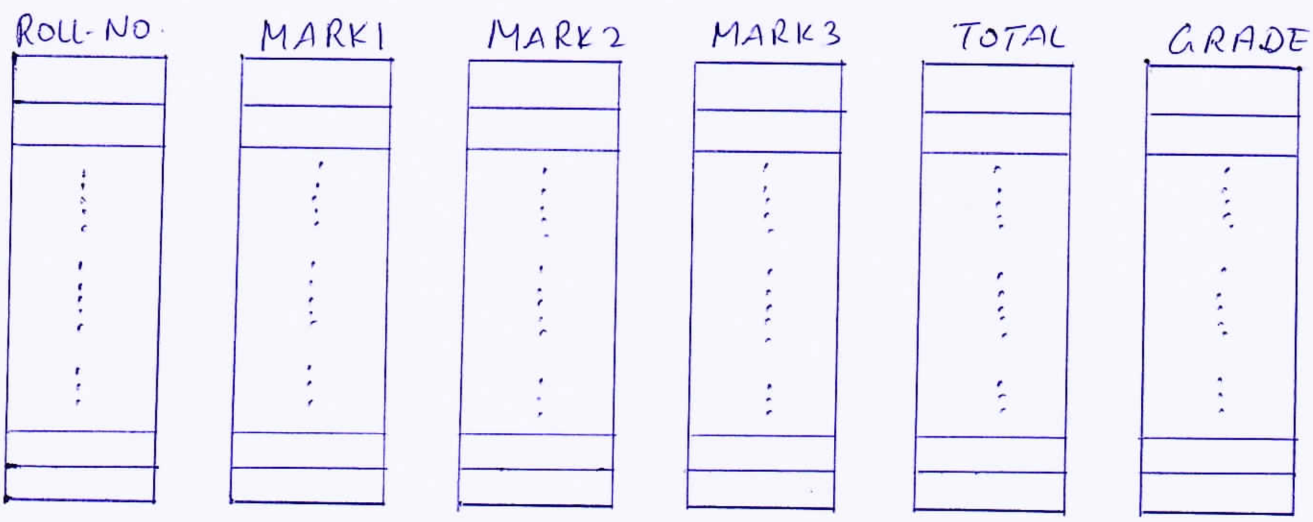

fig: Storing the students records