Chapter 2 Arrays

2.1 What is an Array?

An array is a collection of homogeneous or similar elements. All the elements have same name. To refer to each individual element, index number is used.

For declaring an array, we specify its size and data type

Eg. int arr[5] = {25,35,45,55,65};

Array index	**Arr[0]**	**Arr[1]**	**Arr[2]**	**Arr[3]**	**Arr[4]**
Value	25	35	45	55	65
Memory address	240	242	244	246	248

2.2 Memory Representation of an Array

An array has lower bound, upper bound and size. **Lower bound**: It is the starting index value. It can be 0 or 1. **Upper Bound:** It is the ending index value. It can be Size-1 or Size. If Lower bound is 0 then upper bound is Size-1. If lower bound is 1 then upper bound is Size. Size is the maximum number of elements in the array. It can be calculated as:

Size = (Upper Bound – Lower Bound +1

Why we add 1 here? In mathematics, 20-10 means 10 is not included in the answer. But in array the lower bound is included in the answer, so we add one.

The array is allocated contiguous locations in memory. If the size of the array is 5 and data type of array is integer (sizeof(integer) = 2), then 5X2=10 memory spaces are allocated to the array. The address of each member of the array is calculated as follows:

Address of Each location: It is found by the formula:

Base_address + (index * size_of_data_type)

Array index	**Arr[0]**	**Arr[1]**	**Arr[2]**	**Arr[3]**	**Arr[4]**
Value	25	35	45	55	65
Memory address	240	242	244	246	248
Address Calculation	240+(0*2)	240+(1*2)	240+(2*2)	240+(3*2)	240+(4*2)

2.3 Multi-dimensional Array

An Array can be 1D, 2D, 3D and more. The array discussed above is One-dimensional array. 2D array is one with 2 index variables. 3D array is one with 3 index variables. Take a 1D array to be a line in a page with fixed size and a linear list of characters. A 2D array can be described as a collection of several lines. It makes use of 2 index variables. The first index variable denotes the line number and second index number denotes the specific character in the line. 3D array is a collection of pages consisting of several lines each. It uses three index variables: the first index is page number, second is line number and third is specific character in the line.

This is further explained with an example:

A	B	C	D	R
E	F	G	H	T
I	J	K	L	G
M	N	O	P	B

S	T	U	V	W
W	X	Y	Z	E
I	B	G	J	R
H	K	L	I	F

In the above image two pages are shown. Each page consists of 4 lines. Each line consists of 5 letters. The array is character in nature. It is declared as:

Char arr[2][4][5].

Page number [0]	Line number ∨ | Character number ->	[0]	[1]	[2]	[3]	[4]
	[0]	A	B	C	D	R
	[1]	E	F	G	H	T
	[2]	I	J	K	L	G
	[3]	M	N	O	P	B

In the above diagram the column headings give the character number of a specific entry. Like A is zero character. B is first character. C is 2 character number. The complete line ABCDR has the number [0]. And the complete page has the number [0]. So if we want to refer to first page we write Arr[0]. If we want to refer to first line of first page, we write Arr[0][0]. If we want to refer to first character A of line 1 on page 1, we write Arr[0][0][0].

2.3.1 Address calculation of 2D Array

For explaining the addressing scheme of a 2D array, an example of 2D integer array is taken below. Declaring a 2D array: int arr[2][5]; . This means the array has 2 rows and 5 columns. The contents of the array are: { {10, 15, 20, 5, 24 }, {23, 14, 56, 78,43 }}. The base address of the array is 240.

2.3.1.1 Row major format

In Row major format, the elements of consecutive rows are kept next to each other. In simple words, the first index is row and second index is column. The following table gives the array reference index along with address and contents.

A[0][0]=240 **10**	**A[0][1]=242** **15**	**A[0][2]=244** **20**	**A[0][3]=246** **5**	**A[0][4]=248** **24**
A[1][0]=250 23	A[1][1]=252 14	A[1][2]=254 56	A[1][3]=256 78	A[1][4]=258 43

The Address calculation is performed as below:

Base_address + (row * total_columns* size_of_data_type) + (col * size_of_data_type)

240 + (1 * 5 * 2) + (2 * 2) = 254 = address of a[1][2] i.e., element 56

2.3.1.2 Column major format

In Column major format, the elements of consecutive columns are kept next to each other. In simple words, the first index is column and second index is row. The following table gives the array reference index along with address and contents.

A[0][0]=240	A[0][1]=242
10	23
A[1][0]=244	A[1][1]=246
15	14
A[2][0]=248	A[2][1]=250
20	56
A[3][0]=252	A[3][1]=254
5	78
A[4][0]=256	A[4][1]=258
24	43

The Address calculation is performed as below:

Base_address + (col * total_rows* size_of_data_type) + (row * size_of_data_type)

240 + (2 * 2 * 2) + (1 * 2) = 250 = address of a[2][1] i.e., element 56

2.4 Applications of an Array

The applications of arrays are numerous. Some are listed below:

- 1Dimensional :
  a) **List of elements**: arrays are commonly used to maintain list of elements in memory in a program.
  b) **Binary Tree representation**: The binary tree is represented using an array in programs where the depth of the tree or size of the tree is not huge.
  c) **Stacks representation**: One dimensional array can be used to represent Stacks.
  d) **Queues representation**: It can be used to represent Queue data structure.
- 2Dimensional:
  a) Matrix representation: A 2D matrix representation is the best way to represent a normal matrix. For less populated matrix sparse representation is used.
  b) Graph representation: For representing a graph in memory 2-D matrix is one of the many possible data structures.
- 3Dimensional:
  a) Video memory with multiple frames: The VDU memory is nothing but a collection of 2D pages where each byte represents one pixel on the screen. Generally VDU memory these days keeps the frames ready as VDU pages that are copied into the VDU memory by a fast processor.
  b) **Paging process**: The paging of user process in operating system is also an example of a collection of 2D pages where program executables are divided into pages and loaded in memory as and when need arises.

2.5 Operations on an array

A[0]	**Akshit**
A[1]	Aarush
A[2]	Arnav
A[3]	Devashish
A[4]	Gurpreet

The array data structure is very useful. It must provide for the basic data structure operations to get its maximum benefit. The following operations can be performed on a array. It is not an exhaustive list. But minimum these operations must be allowed.

- Traverse

A[5]	Priyanka
A[6]	Ziaur
A[7]	
A[8]	
A[9]	
The array size=10 elements Number of slots occupied, n=7 Vacant=3	

- Search
- Insert
- Delete
- Sorting

To understand the working of array and the operations on array the following example will be used. IT is a list of students enrolled in data structure course. The list contains just student name and stored as a array.

The declaration is char a[10][15];

It's a character array with 10 names of length 15. The last character will be a '\0' representing the end of array. So the name can be 14 characters long.

2.5.1 Traverse

Array Traversal means accessing individual elements of the array. Since array elements are sequential and can be accessed by index number along with array name the traversal is very simple. E.g. printing all the members of the list or multiplying all the members of the list by 10. The algorithm for Traversal is as follows:

```
Algorithm Traversal ( arr, n)
  • Arr is the input array to be traversed
  • N is the number of elements in the array
For I = 0 To N-1 Step 1
  • Print arr[I]
End for
End
```

The input to the algorithm is the array and number of elements present in the array. Assuming the lower bound is at 0, the array index 0 to N-1 is read for accessing individual elements. A very common application of traversal is to find the maximum or minimum elements in the array. The algorithm for both is given below.

Maximum Element in a List

Minimum Element in a List

Algorithm MaxArray (arr, n)

- Arr is the input array to be traversed
- N is the number of elements in the array

Max=0

For I = 0 To N-1 Step 1

- If arr[I] > Max then Max = arr[I]

End for

Print Max+ “is Largest”

End

Algorithm MinArray (arr, n)

- Arr is the input array to be traversed
- N is the number of elements in the array

Min=arr[0]

For I = 0 To N-1 Step 1

- If arr[I] < Min then Min = arr[I]

End for

Print Min+ “is Smallest”

End

As we can see both the algorithms are similar except that the condition that we test is different. Also in maximum we initialize max with 0 and in minimum with A[0]. This is important and must be made note of. In max all the elements larger than max need to be tracked so 0 will work. But in minimum, we have to compare with a[0] because 0 will be smaller than most of the numbers(except negative numbers). Taking the list of names into consideration the traversal, minimum and maximum program functions will be as follows.

```
void traverse(char a[][15], int n)
{int i=0;
for( i=0; i < n; i++)
printf("%s",a[i]);
}
```

a)

```
void maximum(char a[][15], int n)
{int i=0; char max[15]= '\0';
for( i=0; i < n; i++)
{     if (strcmp( a[i], max)> 1)
                strcpy( max, a[i]);
}
printf("maximum is %s",max);
}
```

b)

```
void minimum(char a[][15], int n)
{int i=0; char min[15]= a[0];
for( i=0; i < n; i++)
{     if (strcmp( a[i], min)<0)
                strcpy( min, a[i]);
}
printf("minimum is %s",min);
}
```

c)

The box (a) above has logic for traversal of the character array of names. It will print all the names in a series. Box (b) contains logic for finding the maximum element. The max is initialized with '\0' (null) character. Whenever a value bigger than this is found, it will be replaced. The program makes use of Strcmp to compare strings. It returns 1 if a[i] is larger, 0 if equal to max and -1 if smaller than max. strcpy is another c language function that is used to copy the a[i] value into max. Box (c) shows the logic for minimum function. It stores the first element of array a into min. Makes use of strcmp and strcpy just like maximum function.

2.5.2 Insert

In array Insertion means to add an element in the existing list. Taking the same list example, suppose we want to add a element 'Devesh' to the list of names. Since the list is sorted in alphabetic order, proper position for the name has to be found and then the insertion is to be made. There are many cases of insertion:

A[0]	**Akshit**
A[1]	Aarush
A[2]	Arnav
A[3]	Devashish
A[4]	Gurpreet
A[5]	Priyanka
A[6]	Ziaur
A[7]	
A[8]	
A[9]	

- Insertion at beginning
- Insertion at end
- Insertion at the location given
- Insertion at sorted Position

Insertion at beginning and end are special cases of Insertion at specified location. Insertion in sorted position requires to compare all the

The array size=10 elements
Number of slots occupied, n=7
Vacant=3
If we want to add the element "Devesh" to the list. First all the names from Gurpreet to Ziaur have to be shifted downward.
Call is Insert(arr, 7, 10, 4, "Devesh")

elements with item to insert and find location. Then perform regular insert with insertion at location.

The algorithm for insertion is given below. It needs the array as input along with number of elements =n and size of the array. Both are different as size is capacity of the array and n is values actually stored. The array arr and the number of element n must be passed in modifiable form or pointer form as both will change value in the algorithm. The list will change due to addition of new name along with the number of element increasing by one. The other two arguments to that are provided are loc and item. The loc is the place where insertion is to be made and item is the value to insert.

Algorithm Insert (arr, n, size loc, item)
- Arr is the input array to be traversed
- N is the number of elements in the array
- Size is the total capacity of the array
- Loc is the location where insert is to be made
- Item is the element to be searched

If loc > size or loc > n+1
- Print overflow

For I = N-1 To Loc Step -1
- arr[I +1]= arr[I]

End for

Arr[loc] = item

N=n+1

End

The algorithm is divided into three parts:

- Check overflow
- Shift elements down
- Make the insertion and increase number of elements.

First loc is compared with size and n. If loc is more than size that means we are trying to modify a location that does not exist. If location is more than n+1 then we are trying to insert outside the element of the array. In both the cases overflow is to be reported. Second, if overflow is not there then shift the elements down by one position from n-1 to loc (if lower bound is zero). Finally, copy the new element at arr[loc]. Increment the value of n by one.

The code for making an insertion in the given list of names is as follows:

```
void insert( char a[][15], int *n, int size, int loc, char item[15])
{
        int i=0;
         if( loc <size && loc <= *n+1)
         {
             for( i =*n-1, i>=loc; i--) strcpy(a[i+1], a[i]);
            strcpy(a[loc], item);
             *n= *n +1;
```

```
        }
}
```

2.5.3 Delete

The delete operations is to remove an element from the list or array. It can be summarized as three steps:

- Check underflow. That is to check if array is empty or the item we are trying to delete is not present in the array.
- Shift elements up for deletion
- Decrement the value of n by one for deleted item.

```
Algorithm Delete ( arr, n, size loc)
•Arr is the input array to be traversed
•N is the number of elements in the array
•Size is the total capacity of the array
•Loc is the location where insert is to be made
If loc > size or loc > n
•Print underflow
For I = loc To N-1  Step 1
•arr[ I ]= arr[ I +1 ]
End for
n=n-1
End
```

The algorithm for delete is given below. It accepts four arguments. Arr to represent the array to update. N is the number of elements actually present in the array. Size if the capacity of the array and last the location of item to delete. First check for underflow that if loc is more than size or n then we are trying to delete an item that does not exist. The shift the elements up by one place in next step. Lastly decrement the value of n by 1.

Like insertion, deletion can be made at beginning, end, at a location or in sorted order. The one discussed here is that of deleting at a given location. Deleting at end and beginning are special cases of this. For deleting in sorted order, find the ,loc of item and then call the deletion by location.

The array size=10 elements
Number of slots occupied, n=7, Vacant=3
If we want to remove the element “Gurpreet” then both the names below it are shifted upwards.
Call is Delete(arr, 7, 10, 4)

```
void insert( char a[][15], int *n, int size, int loc, char item[15])
{    int i=0;
    if( loc <size && loc <= *n)
    {     for( i =loc, i<*n-1; i++) strcpy(a[i], a[i+1]);
          *n= *n -1;
    }
}
```

2.5.4 Searching

Search operation is most basic operation that needs to be performed on a data structure. After the data is stored, there is a need to find it and process it in the required manner. Be

it an in-memory data structure or secondary storage data structure, searching is a basic operation. For in-memory data structure several searching methods exist. The two most popular are- Linear Search and Binary Search, discussed below.

2.5.4.1 Linear Search

In linear search, each item is visited once and matched with value to be searched. If a match is found then success is reported otherwise if after matching whole list we cannot find the element, then failure is reported. Linear search does not expect any special characteristic from the input list.

The elements are arranged in a list with no mutual ordering. The elements can appear in any order and any arrangement. The list is linear. The item to be searched is matched with each entry in the list till a match is found. If match is found then that location is the answer. Otherwise if in whole list item is not found then failure is returned.

Eg. We are given a list: 5,2,4,9,8,13,11,45,78,90,54,3.

The item to search is 45. First we match the item with each list member in sequential order. The element 45 is found at location 8 (if first element is at location=1). If we try to find item=39. The item is compared with all the elements in the list and not found in the list. So failure is reported.

```
Algorithm Search ( arr, n, item)
  • Arr is the input array to be traversed
  • N is the number of elements in the array
  • Item is the element to be searched
Found =False
For I = 0 To N-1 Step 1
  • If item == arr[I] then {Found = True,      break;}
End for
If Found== True { print item found at I; return I; }
Else { print item not on list; return -1; }
End
```

As can be seen in the algorithm given in the left. The input variables are: list, size of list-n and item to be searched. The output is location of element. Its value is -1 if item is not found. Otherwise the location is index number where the item is found. In for loop the items are iterated in linear order from 1 to n and item is matched with list[i]. If a match is found then break out of the loop.

The **time complexity** of linear search is n. If there are n elements in the list then atmost n comparisons need to be performed.

The array size=10 elements
Number of slots occupied, n=7
Vacant=3
If we search for the value Devashish, it will be reported at index value: 3 and location 4.
IF we search for Devesh, item not found is printed.

Best case: Finding the element at index=0. That is item to search is the first element in the list. It takes 1 operation so complexity in best case is O(1).

Worst case: Finding the element at index=n-1. That is item to search is the last element in the list. It takes n operation so complexity in worst case is O(N).

Average case: The probability of finding the element is calculated. There is an equal chance that element can be found at first location or second or third or n location.

Adding up all these cases we get:

(1+2+3+…….+n) / n

Since there are n cases so divide the above value by n.

n(n+1)/2n = (n+1)/2

2.6.4.2 Binary Search

Drawbacks of Linear Search: Average and worst case complexity is O(N) which is very time consuming. We need a method with better search time as in linear search we have to traverse all the elements before item can be found. Another way of doing it is to arrange the elements in sorted order and take the benefit of the sorted list.

Algorithm IterativeBinarySearch(item, list, n)

- **Input variables: list is the array that contains the list of elements sorted in ascending order.**
- **item is the value to be searched.**
- **n is the size of list**
- **Output: Location where the element is found.**

location=-1

while(beg<=end)

- **mid = (beg+end)/2**
- **if (item==list[mid) location=mid**
- **else if (item < a[mid]) end= mid-1**
- **else if (item > a[mid]) beg = mid +1**

end for

return location

Binary search reduces the cost of search operation to log n. Instead of searching all n elements in the list, selective matching is

performed. **The elements are assumed to be arranged in ascending (or descending) order**. The array begins at beg and ends at end. Calculate mid= (beg + end) / 2. First we match the elements at mid i.e. a[mid] with item. If match is found we return the location as answer. Otherwise if item is smaller than a[mid](item<a[mid]) then update end =mid-1. Otherwise if item> a[mid], set beg=mid+1.

eg. the list is 5,2,4,9,8,13,11,45,78,90,54,3 and the item to search is 45.

arrange the elements in ascending order: 2,3,4,5,8,9,11,13, 45,54,78,90. Beg=1 and end=12.

a) mid= (1+12)/2 = 13/2= 6.5 = take only integer part=6.

compare item with a[6], 45 == 9? no, 45 < 9? no, 45>9 ?yes, so update beg=7.

mid= (7+12)/2 = 19/2= 9.5 = take integer part = 9.

compare item with a[9], 45 == 45? yes. Stop search as item is found.

b) If we try to find item=39. Beg=1 and end=12.

mid= (1+12)/2 = 13/2= 6.5 = take only integer part=6.

compare item with a[6], 39 == 9? no, 39 < 9? no, 39 > 9 ? yes, so update beg=7.

mid= (7+12)/2 = 19/2= 9.5 = take integer part = 9.

compare item with a[9], 39 == 45? no, 39 < 45? yes, update end=8.

mid= (7+8)/2 = 15/2= 7.5 = take integer part = 7.

compare item with a[7], 39 == 11? no, 39 < 11? no, 39 > 11, yes, update beg=8.

beg=end=8, mid=8.

Compare the item 39 with a[8]. Match not found so we stop and return failure.

Eg. You have a list of students arranged in alphabetic order. If you are looking for a name beginning with ‘N’, we do not start from ‘a’ instead we directly jump to the portion where names with N appear and search there.

To Search for a number in a list of sorted values.

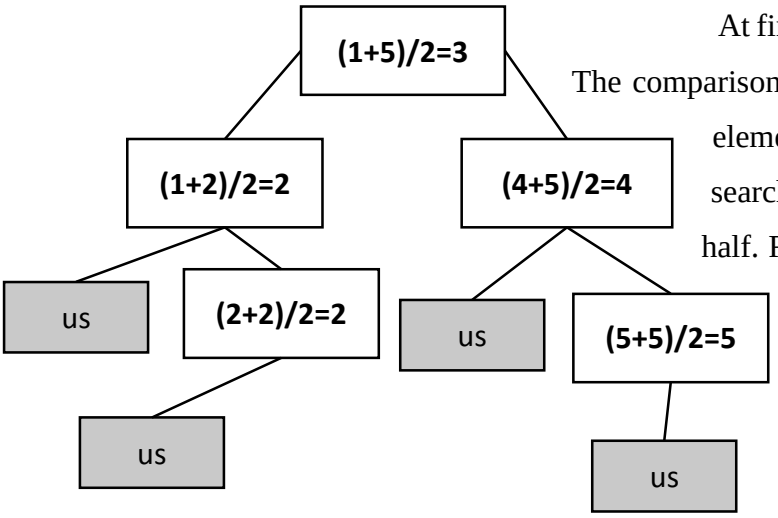
Similarly Binary search looks for the element at middle location of the array. IF match is found then it is reported. Otherwise calculate new mid point.

```
Algorithm RecursiveBinSrch (a, l, u, x)
• // Given an array a [l:u]of elements in non-decreasing
• // Order, 1 <= l <=u, determine whether x is present, and
• // if so , return j such that x=a[j] ; else return 0.
if (l=u) then // if Small(p)
•If (x=a[l]) then return l;
•else return 0;
else //Reduce P into a smaller subproblem.
•mid := [ (l+u) / 2] ;     // problem of divide by zero
•if( x = a[mid] ) then return mid ;
•else if(x < a[mid] ) then     return Binsrch( a , l ,mid-1 , x) ;
•else return BinSrch( a, mid+1 , u ,x)
```

3. small(P) will return value **i** if **x=a(i) o**therwise zero.

Example:

45	55	109	122	135
1	2	3	4	5



Formal definition of the problem:

- Let a(i) , 1<=i<=n, be a list of elements that are sorted in non-decreasing order.
- Problem P is to find if an element x is contained in the list, if it is we have to find the location of x.

1. Let P=(n, a(i),…..a(l),x)
2. if n=1, Then small(P)

At first the mid is calculated at (1+5)/2= 3. The comparison is made at element at location 3. If element is smaller than item to search we search only in lower half otherwise upper half. Repeatedly we find the mid till either Unsuccessful search (US) is reached or item is found in the list.

2.6.4.3 Other Search Methods

Ternary Search

In this while calculating mid the beg+end is divided by 3 instead of 2. This further makes the search process faster. The elements are sorted in Ascending order in the list.

Interpolation Search

The formula for calculation of mid is:

mid= beg + ((end- beg) / (a[end] - a[beg])) * (item - A[beg])

The elements are sorted in ascending order in the list.

Hash Table

In all the search methods the searching is performed in sequential fashion based on location. In hash table the content is directly addressable by the content itself. The data to store is converted in the address by some formula and accordingly the data is stored. Then for fetching the data the content is directly converted into the address and fetched. This will be more clear from the following example.

Say we have the following data to store: 5,2,4,9,8,13,11,45,78,90,54,3

We have a array of 10 elements to store the data. So size=10.

The location of the element to store is found using division method : divide the item with size. So location of 5 is 5%10 = 5 (remainder). location of 2 is 2%10=2, location of 4 is 4%10 = 4. Location of 78 is 78%10 = 8.

Now the question arises that two elements can have same location by this method. To deal with colliding locations three methods are there: Chaining, linear probe and quadratic probe. In chaining the elements with same location are stored as a linked list. In linear probe the next location+1 place is used to store the elements with colliding address. In quadratic probe ((2Xlocation) +1) is used to store the elements with colliding location. Then location +2, location +3 etc are used. Now question arises that if an element actually has a value of location+3 and that is already occupied by a colliding element then where it is stored? It is treated like other elements and one of the three methods may be used. Only one method is to be used in one hash table.

2.6.4.4 Comparison of Linear and Binary Search

Criteria	Linear Search	Binary Search
Definition	It searched for the item in complete list till the item is found.	It searches for the item in a part of the list and can find if item is there or not.
Assumption	No assumption on nature of list	Assumes the list to be sorted in ascending or descending order.
Logical Complexity	Very simple as only comparison to be made with all the elements sequentially.	Not very easy as comparison progresses depending on whether the item is small, large or equal to the value in the list.

Time Complexity	Takes as many operations as the size of the list. O(n) is the complexity	Takes Log(n) operations. Involves extra effort in sorting the list.
Nature of Comparisons	Only equality is checked	Ordering comparisons
Speed	Linear Search is slow	Binary Search is fast
Technique	Sequential	Divide and Conquer
Data Structure	Can be implemented on array as well as linked list	Difficult to implement on Linked List
Size of data	Unsuitable for large data lists	Suitable for large data lists.

2.6.5 Sorting

There are five common sorting methods as stated below. They are discussed in next chapter.

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort