DATA STRUCTURES

UNIT 1

BASIC TERMINOLOGIES OF DATA STRUCTURES

Data : Data can be defined as an elementary value or the collection of values or set of values.
For Example : student name and student id are the data about the student.

Data items : Data items refers to a single unit of values.
For Example : student name

Group items : Data items that are divided into sub-items are called Group items.
For Example : A Student Name may be divided into three subitems- first name, middle name, and last name.

Elementary Items : Data items that are not able to divide into sub-items are called Elementary items.
For Example : Student's id

Entity : An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric.
For Example : Attributes - Names, Age, Sex, Student-id
Values- Arun, 19, M, IT2022001
Entities with similar attributes form an entity set. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute.
The term "information" is sometimes used for data with given attributes or in other words meaningful processed data.

Record : Record can be defined as the collection of various data items.
For Example : if we talk about the student entity, then its name, id. address, course and marks can be grouped together to form the record for the student.

File : A File is a collection of various records of one type of entity.
For Example : if there are 60 students in the class, then there will be 60 records in the related file where each record contains the data about each student.

Field : Field is a single elementary unit of information representing the attribute of an entity.

Each record in a file may contain many field items but the value in a certain field may uniquely determine the record in the file. Such a field K is called a primary key and the values k1, k2, ….. kn in such a field are called keys or key values.
Records may also be classified according to length. A file can have fixed-length records or variable-length records.
In fixed-length records, all the records contain the same data items with the same amount of space assigned to each data item.
In variable-length records, file records may contain different lengths.

For Example : Student records have variable lengths, since different students take different numbers of courses. Variable-length records have a minimum and a maximum length.

The above organization of data into fields, records and files may be complex enough to maintain and efficiently process certain collections of data. For this reason, data are also organized into more complex types of structures.

WHY DATA STRUCTURES

Processor Speed : To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search : Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple Requests : If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

DATA STRUCTURE DEFINITION

Data Structure is a method, technique and tool to perform operations on data.

Or

A data structure is logical way of storing and organizing data either in computer's memory or on the disk storage so that it can be used efficiently.

Selecting a data structure as follows:
1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally classified into
1. Primitive data Structures
2. Non-primitive data Structures

1. Primitive Data Structures : Primitive data structures are the fundamental data types which are supported by a programming language or Data Structures that are directly operated upon the machine-level instructions.
Basic data types such as integer, float, character and Boolean are known as Primitive data Structures. These data types consist of characters that cannot be divided and hence they also called simple data types.

2. Non- Primitive Data Structures : Non-primitive data structures are those data structures which are created using primitive data structures. Examples of non-primitive data structures is the processing of complex numbers, linked lists, stacks, trees, and graphs.

Based on the structure and arrangement of data, non-primitive data structures are further classified into

1. Linear Data Structure
2. Non-linear Data Structure

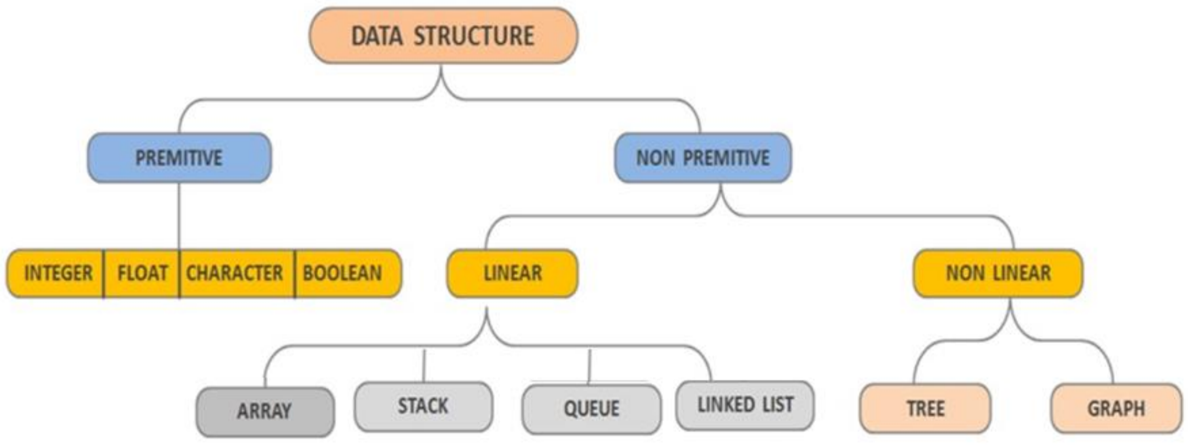
1. Linear Data Structure :
A data structure is said to be linear, if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.

(i) One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.

(ii) The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The common examples of linear data structure are Arrays, Stacks, Queues and Linked lists

2. Non-linear Data Structure :
A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear data structure.



OPERATIONS ON DATA STRUCTURE

Insertion : Insertion can be defined as the process of adding the elements to the data structure at any location. If the size of data structure is n then we can only insert n-1 data elements into it.

Deletion : The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

Traversing : Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

For Example : If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

ALGORITHM

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer.

Definition : An algorithm is the finite set of instructions which are being carried in a specific order to perform the specific task.
It is not the complete program or code; it is just a set of instructions for solving a problem or accomplishing a task.

Analysis of an Algorithm
The analysis of an algorithm is a technique that measures the performance of an algorithm.

Why - Algorithm Analysis
1. To predict the behavior of an algorithm without implementing it on a specific computer.
2. It is much more convenient to have simple measures for the efficiency of an algorithm than to implement and test.
3. By analyzing different algorithms, one can compare them to determine the best one.

The factors over which the algorithms majorly depend are the space (Memory usage) and time (CPU Time) required to execute it.
Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity, or volume of memory, known as space complexity.
The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.

Characteristics or properties of an algorithm:
1. The instructions must be in an ordered form.
2. The instruction must be simple and concise.
3. There must be no ambiguity(doubt) in any instruction
4. It should conclude after a finite number of steps.
5. Repetitive programming constructs must possess an exit condition, otherwise the program might run infinitely
6. The algorithm must be complete and solve the given problem.

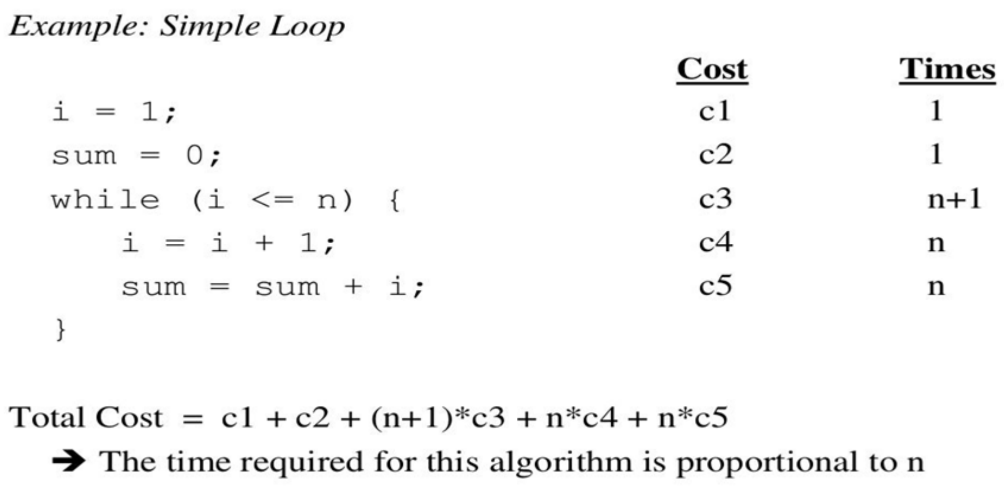
Qualities of a good algorithm:
1. It should use the most efficient logic to solve the given problem. (time complexity)
2. It should use minimal system memory for its execution. (space complexity)
3. It should generate the most accurate results
4. It should be easy to implement in the form of a program
5. It should be designed with standard conventions so that others are able to easily modify it while adding additional functionality

TIME COMPLEXITY

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean
- the number of memory accesses performed
- the number of comparisons between integers
- the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

Frequency count
It specifies the number of times the statement is to be executed.
For comments, declaration – step count = 1
Return and assignment – step count = 1
while (<expr>) do – step count = the number of times <expr> is executed.
for i=<expr> to <expr1> do – step count = number of times <expr1> is checked.
Ignore low order exponents
For Example : $3n$^4+$4n$^3+$10n$^2+n+100 =O(n^4)

Example : 1

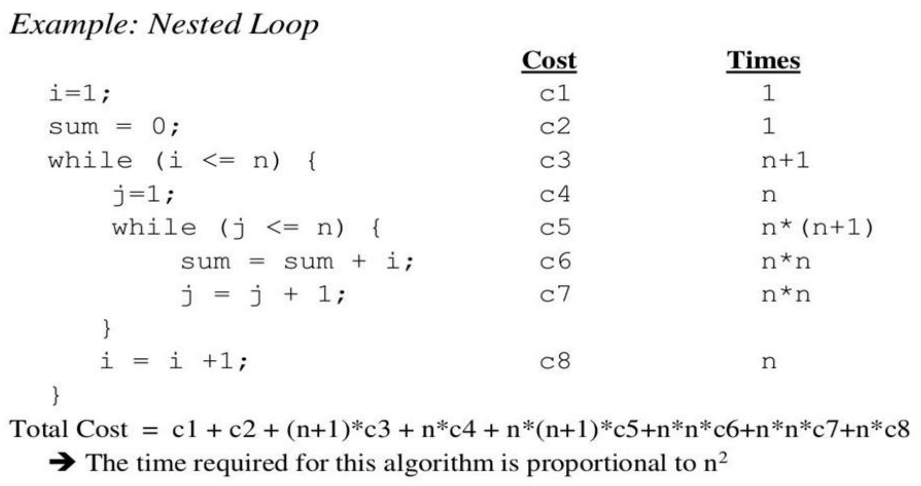


Example: Simple Loop

	Cost	**Times**
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5
➔ The time required for this algorithm is proportional to n

Example : 2



Example: Nested Loop

	Cost	**Times**
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`    i = i +1;`	c8	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8
➔ The time required for this algorithm is proportional to n^2

SPACE COMPLEXITY :

Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this. We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc.
In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

Example : 1

```
Algorithm abc (a, b, c)
{
  return a+b+b*c+(a+b-
c)/(a+b)+4.0;
}
```

For every instance, 3 computer words required to store variables: a, b, and c. Therefore

$$S_p() = 3. \; S(P) = 3.$$

Example : 2

```
Algorithm Sum(a[ ], n)
{
      s:= 0.0;
      for i = 1 to n do
            s := s + a[ i ];
      return s;
}
```

Every instance needs to store array a[ ] & n.
Space needed to store n = 1 word.
Space needed to store a[ ] = n floating point words (or at least n words)
Space needed to store i and s = 2 words
Sp(n) = (n + 3).
Hence S(P) = (n + 3).

ASYMPTOTIC NOTATIONS

It is the mathematical way of representing the time complexity of an algorithm for analysis. In general, it tells us about how good an algorithm performs when compared to another algorithm.
The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

Big Oh (O)	– Big O Notation – provides an upper bound for the function f
Omega (Ω)	– Omega Notation – provides a lower-bound for the function f
Theta (Q)	– Theta Notation – is used when an algorithm can be bounded both from above and below by the same function f

Usually, time required by an algorithm falls under three types:

Best Case	− Minimum time required for program execution.
Average Case	− Average time required for program execution.
Worst Case	− Maximum time required for program execution.

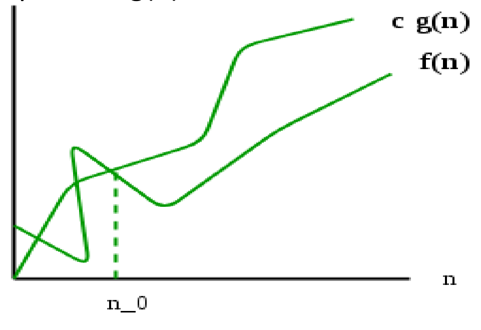
Asymptotic analysis is input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

For Example : running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases.

Big O Notation :

It represents the upper bound running time complexity of an algorithm. It always indicates the maximum time required by an algorithm for all input values. It describes the worst case of an algorithm time complexity and it is the most commonly used notation.

Big - Oh Notation can be defined as follows...
Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term.
f(n) = O(g(n)) iff positive constants c and n0 exist such that
f(n) <= C g(n) for all n >= n0, C > 0 and n0 >= 1.
Then we can represent f(n) as O(g(n)).
f(n) = O(g(n))
(i.e.), O(g) comprises all functions f, for which there exists a constant c and a number n0, such that f(n) is smaller or equal to c·g(n) for all n, n ≥ n0.



f(n) = O(g(n))

Time complexity of an algorithm using O Notation:

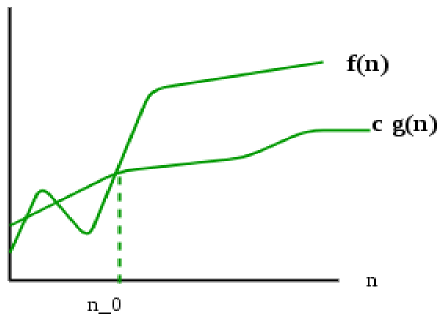
Running TIME	**NAME**
O(1)	constant
O(n)	linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$,$O(3^n)$, $O(k^n)$	Exponential
$O(n^k)$	Polynomial
O(log n)	Logarithmic
O(n log n)	Log linear

Some of the commonly occurring time complexities in their ascending orders of magnitude are listed as: O(1) ≤ O(log n) ≤ O(n) ≤ O(n log n) ≤ O(n2) ≤ O(n3) ≤ O(2n)

Omega (Ω) Notation :

It represents the lower bound running time complexity of an algorithm. It always indicates the minimum time required by an algorithm for all input values. It describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...
Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term.
f(n) = Ω(g(n)) iff positive constants c and n0 exist such that
f(n) >= C g(n) for all n >= n0, C > 0 and n0 >= 1.
Then we can represent f(n) as Ω(g(n)).
f(n) = Ω(g(n))
(i.e.) Ω(g) comprises all functions f, for which there exists a constant c and a number n0, such that f(n) is greater or equal to cg(n) for all n ≥ n0.



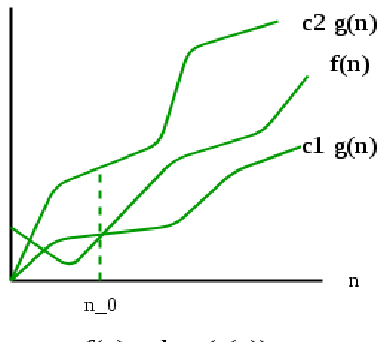
f(n) = Omega(g(n))

Theta (Q) Notation

It is used to define the average bound of an algorithm in terms of Time Complexity. It always indicates the average time required by an algorithm for all input values. It describes the average case of an algorithm time complexity.

Dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved.

Big - Theta Notation can be defined as follows...
Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term.
f(n) = Q(g(n)) iff positive constants c1, c2 and n0 exist such that
C1 g(n) <= f(n) <= C2 g(n) for all n >= n0, C1 > 0, C2 > 0 and n0 >= 1.
Then we can represent f(n) as Θ(g(n)).
f(n) = Θ(g(n))
(i.e.), f lies between c_1 times the function g and c_2 times the function g, except possibly when n is smaller than n_0.



f(n) = theta(g(n))

TIME-SPACE TRADE OFF

In computer science, a space-time or time-memory tradeoff is a way of solving a problem in :
1) Either in less time and by using more space or
2) By solving a problem in very little space by spending a long time.

Types of Trade Off:
1. Compressed / Uncompressed Data
2. Re-Rendering / Stored Images
3. Smaller Code / Loop Unrolling
4. Lookup Table / Recalculation

1. Compressed / Uncompressed Data
• A space–time tradeoff can be applied to the problem of data storage.
• If data is stored uncompressed, it takes more space but access takes less time than if the data were stored compressed
• Since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm
• Depending on the particular instance of the problem, either way is practical.
• There are also rare instances where it is possible to directly work with compressed data, such as in the case of compressed bitmap indices, where it is faster to work with compression than without compression.

2. Re-Rendering / Stored Images
• Storing only the SVG source of a vector image and rendering it as a bitmap image every time the page is requested would be trading time for space; more time used, but less space.

• Rendering the image when the page is changed and storing the rendered images would be trading space for time; more space used, but less time. This technique is more generally known as caching.

3. Smaller Code (with loop) / Larger Code (without loop)

• Smaller code occupies less space in memory but it requires high computation time which is required for jumping back to the beginning of the loop at the end of each iteration.
• Larger code or Loop unrolling can optimize execution speed at the cost of increased binary size. It occupies more space in memory but requires less computation time. (as we need not perform jumps back and forth since there is no loop.)
• Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program.
• We basically remove or reduce iterations.
• Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

This program does not uses loop unrolling.

```
#include<stdio.h>
int main(void)
{
        for (int i=0; i<5; i++)
                printf("Hello\n"); //print hello 5 times

        return 0;
}
```

```
// This program uses loop unrolling.
#include<stdio.h>
int main(void)
{
   // unrolled the for loop in program 1
   printf("Hello\n");
   printf("Hello\n");
   printf("Hello\n");
   printf("Hello\n");
   printf("Hello\n");
   return 0;
}
```

Advantages

• Increases program efficiency.
• Reduces loop overhead.
• If statements in loop are not dependent on each other, they can be executed in parallel.

Disadvantages

• Increased program code size, which can be undesirable.
• Possible increased usage of register in a single iteration to store temporary variables which may reduce performance.

• Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions.

4. Lookup Table / Recalculation
• In lookup table, an implementation can include the entire table which reduces computing time but increases the amount of memory needed.
• A lookup table pre-stores the value of a function that would otherwise be computed each time it is needed.
For example, 12! is the greatest value for the factorial function that can be stored in a 32-bit int variable.
• It can recalculate i.e. compute table entries as needed, increasing computing time but reducing memory requirements.
• A lookup table or lookup file holds static data and is used to look up a secondary value based on a primary value.
• It can be used to translate encoded information into a user-friendly description, to validate input values by matching
them to a list of valid items, or to translate a shorthand entry into something more detailed.
• A lookup action retrieves values from a related table. Specify the starting item, where to locate its value, and the column whose value is to be retrieved.
• For example, using the following table, if you want to display the name of a sales manager for a particular sales team, specify the name of the sales team as the primary value, for example, Rhode Island, and the lookup table returns the name of the sales manager, Mike Lucas, as the secondary value.

Typically, a lookup table is one of the following file types:
• Text file (*.txt)
• Lookup table (*.tbl)
• Tab delimited file (*.tab)
• Lookup table (*.lup)

ARRAYS

An Array is defined as, an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations.
The data items of an array are of same type and each data items can be accessed using the same name but different index value.
An array is a set of pairs, <index, value >, such that each index has a value associated with it. It can be called as corresponding or a mapping

For Example : <index, value>
< 0 , 25 > list[0]=25
< 1 , 15 > list[1]=15
< 2 , 20 > list[2]=20
< 3 , 17 > list[3]=17
< 4 , 35 > list[4]=35

Here, list is the name of array. By using, list [0] to list [4] the data items in list can be accessed.

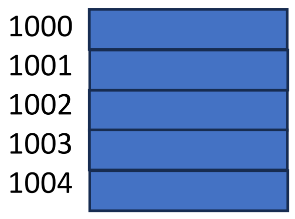
REPRESENTATION OF LINEAR ARRAYS IN MEMORY

A linear array is a list of a finite number ‘n’ of homogeneous data element such that
- The elements of the array are reference respectively by an index set consisting of n consecutive numbers.
- The element of the array are respectively in successive memory locations.

The number n of elements is called the length or size of the array. The length or the numbers of elements of the array can be obtained from the index set by the formula
When LB = 0,
Length = UB – LB + 1
When LB = 1,
Length = UB

Where, UB is the largest index called the Upper Bound
LB is the smallest index, called the Lower Bound

Representation of linear arrays in memory
Let LA be a linear array in the memory of the computer. The memory of the computer is simply a sequence of address location as shown below,



LOC (LA [K]) = address of the element LA [K] of the array LA
The elements of LA are stored in successive memory cells.
The computer does not keep track of the address of every element of LA, but needs to keep track only the address of the first element of LA denoted by,
Base (LA) and called the base address of LA.

Using the base address of LA, the computer calculates the address of any element of LA by the formula
LOC (LA[K]) = Base(LA) + w(K – lower bound)
Where, w is the number of words per memory cell for the array LA.

TYPES OF ARRAYS :

1. One Dimensional or Single Dimensional Arrays
2. Two Dimensional Arrays
3. Multi-Dimensional or 'n' Dimensional Arrays
4. Static array - the compiler determines how memory will be allocated for the array
5. Dynamic array - memory allocation takes place during execution

One-Dimensional Arrays :
The simplest type of arrays, one-dimensional arrays, contains a single row of elements. These arrays are usually indexed from 0 to n-1, where ‘n’ is the size of the array. Utilizing its assigned index number, each element in an array can be conveniently accessed.

Two-Dimensional Arrays :
Two-dimensional array type are arrays that contain arrays of elements. These are also referred to as matrix arrays since they can be thought of as a grid that lays out the elements into rows and columns. Each element within the two-dimensional array can be accessed individually by its row and column location. This array type is useful for storing data such as tables or pictures, where each element may have multiple associated values.

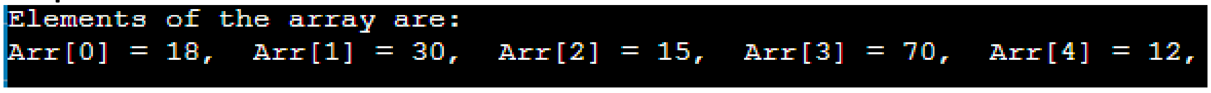
Multi-Dimensional Arrays :
Multi-Dimensional arrays are a powerful data structure used to store and manage data organizationally. This type of arrays consists of multiple arrays that are arranged hierarchically. They can have any number of dimensions, the most common being two dimensions (rows and columns), but three or more dimensions may also be used.

Basic operations of Array :

Traversal – The array's elements are printed using this operation.

```
#include<stdio.h>
void main()
   {
     int Arr[5] = {18, 30, 15, 70, 12};
     int i;
     printf("Elements of the array are:\n");
     for(i = 0; i<5; i++)
     {
       printf("Arr[%d] = %d,  ", i, Arr[i]);
     }
   }
```

Output



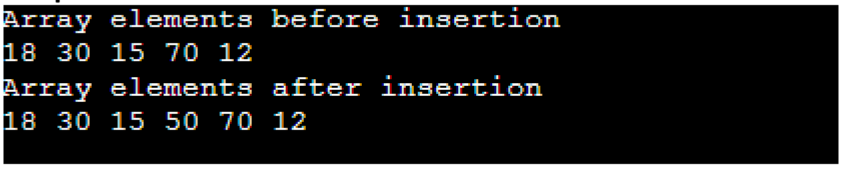
```
Elements of the array are:
Arr[0] = 18,  Arr[1] = 30,  Arr[2] = 15,  Arr[3] = 70,  Arr[4] = 12,
```

Insertion – It's used to add an element to a specific index.

```
#include <stdio.h>
int main()
{
   int arr[20] = { 18, 30, 15, 70, 12 };
   int i, x, pos, n = 5;
   printf("Array elements before insertion\n");
```

```
        for (i = 0; i < n; i++)
            printf("%d ", arr[i]);
            printf("\n");

        x = 50; // element to be inserted
        pos = 4;
        n++;

        for (i = n-1; i >= pos; i--)
        arr[i] = arr[i - 1];
        arr[pos - 1] = x;
        printf("Array elements after insertion\n");
        for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
        printf("\n");
        return 0;
    }
```

Output

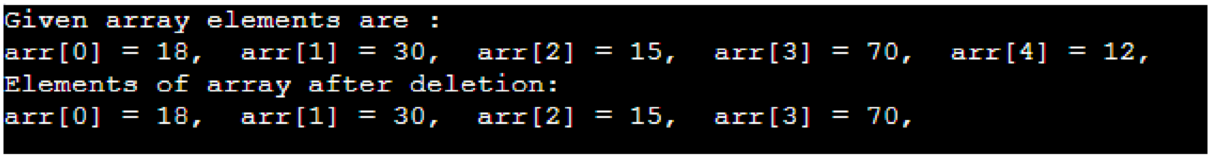


```
Array elements before insertion
18 30 15 70 12
Array elements after insertion
18 30 15 50 70 12
```

Deletion – It is used to remove an element from a specific index.

```
    #include <stdio.h>
    void main() {
      int arr[] = {18, 30, 15, 70, 12};
      int k = 30, n = 5;
      int i, j;

      printf("Given array elements are :\n");

      for(i = 0; i<n; i++) {
        printf("arr[%d] = %d,  ", i, arr[i]);
      }

      j = k;

      while( j < n) {
        arr[j-1] = arr[j];
        j = j + 1;
      }

      n = n -1;

      printf("\nElements of array after deletion:\n");

      for(i = 0; i<n; i++) {
        printf("arr[%d] = %d,  ", i, arr[i]);
      }
    }
```

Output



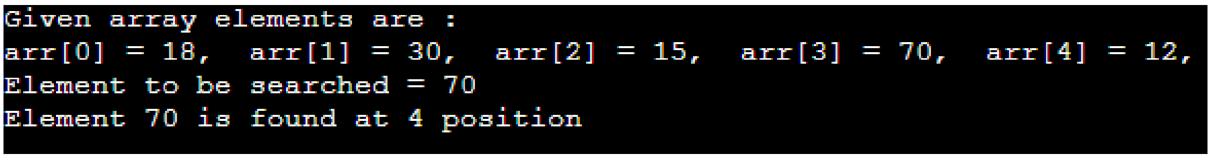
```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Elements of array after deletion:
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,
```

Search – It is used to search for an element using either the value or the specified index.

```
     #include <stdio.h>

     void main() {
        int arr[5] = {18, 30, 15, 70, 12};
        int item = 70, i, j=0 ;

        printf("Given array elements are :\n");

        for(i = 0; i<5; i++) {
           printf("arr[%d] = %d,  ", i, arr[i]);
        }
         printf("\nElement to be searched = %d", item);
        while( j < 5){
           if( arr[j] == item ) {
              break;
           }

           j = j + 1;
        }
```

```
    printf("\nElement %d is found at %d position", item, j+1);
  }
```

Output

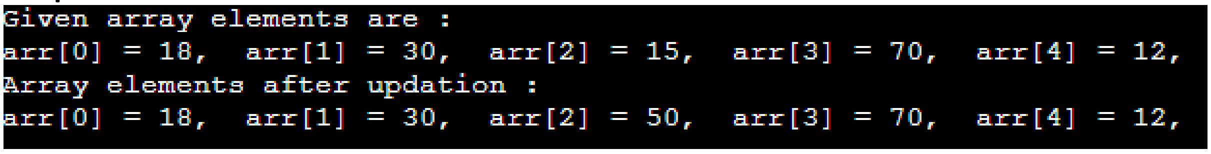


```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Element to be searched = 70
Element 70 is found at 4 position
```

Update – This operation updates an element at a specific index.

```
#include <stdio.h>

void main() {
  int arr[5] = {18, 30, 15, 70, 12};
  int item = 50, i, pos = 3;

  printf("Given array elements are :\n");

  for(i = 0; i<5; i++) {
    printf("arr[%d] = %d,  ", i, arr[i]);
  }

arr[pos-1] = item;
  printf("\nArray elements after updation :\n");

  for(i = 0; i<5; i++) {
    printf("arr[%d] = %d,  ", i, arr[i]);
  }
}
```

Output



```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Array elements after updation :
arr[0] = 18,  arr[1] = 30,  arr[2] = 50,  arr[3] = 70,  arr[4] = 12,
```

Complexity of Array operations

Time and space complexity of various array operations are described in the following table.

Time Complexity

Operation	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Space Complexity

In array, space complexity for worst case is O(n).

SEARCHING

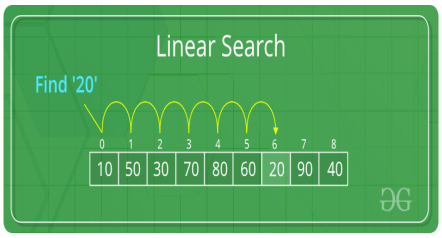
A search in any data structure that allows the efficient retrieval of specific items from a set of items, such as a specific record from a database. The simplest, most general, and least efficient search structure is merely an unordered sequential list of all the items.

Types of Searching

1. Linear search
2. Binary search

Linear Search

1. Linear search also referred as sequential search, is the simplest search technique.
2. The search begins at one end of the test and searches for the required element one by one until the element is found or till the end of the list is reached.
3. The search is said to be successful if the search element is found and unsuccessful if the search element is not found in the list.
4. The linear search technique doesn’t require that the data items in the list are to be in sorted order.



Linear Search Algorithm

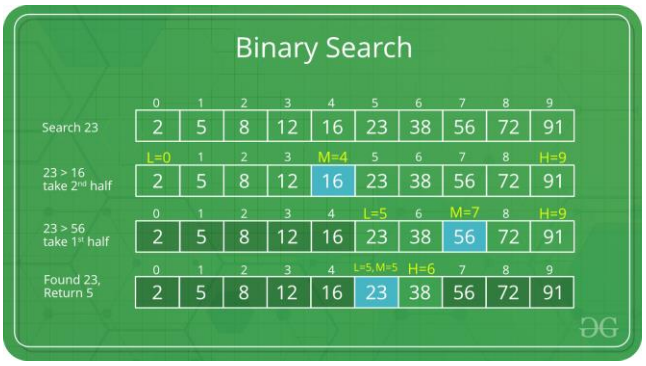
```
Procedure SEQSRCH(F,n,i,K)
//Search a file F with key values $K_1$, ...,$K_n$ for a record $R_i$ such that $K_i$ = K. If there is no such
record, *i* is set to 0//
  $K_0$ <-- K; i <-- n
  while $K_i$ != K do
  i <-- i - 1
 end
end SEQSRCH
```

This algorithm requires *O(n)* key comparisons.
Time required by sequential search is *O(n)*

The drawback of sequential search algorithm is having to traverse the entire list, O(n)
Sorting the list does minimize the time of traversing the whole data set, but we can improve the searching efficiency by using the Binary Search algorithm.

BINARY SEARCH

1. The binary search algorithm is one of the most efficient searching techniques, which requires the list to be sorted in ascending order.
2. To search for an element in the list, the binary search algorithm splits the list and locates the middle element of the list. It is then compared with the search element.
3. If the search element is less than the middle element, the first part of the list is searched else the second part of the list is searched.
4. The algorithm again reduces the list into two halves locate the middle element and compares with the search element. If the search element is less than the middle element the first part of the list is searched.
5. This process continues until the search element is equal to the middle element or the list consists of only one element that is not equal to the search element.



Binary Search Algorithm:

```
procedure  BINSRCH(F, n, i, k)
//Search an ordered sequential file F with records R1, ...,Rn and the keys
K1 <= K2 <= ... <= Kn for a record Ri such that Ki = K; i = 0 if there is no such record else Ki = K.
Throughout the algorithm, l is the smallest index such that Kl may be K and u the largest
index such that Ku may be K//
  l <-- 1; u <-- n
  while l <= u do
  m <-- [(l + u)/2] //compute index of middle record//
  case
  :K > $K_m$: l <-- m + 1 //look in upper half//
  :K = $K_m$: i <-- m; return
  :K < $K_m$: u <-- m - 1 //look in lower half//
  end
  end i <-- 0 //no record with key K//
end BINSRCH
```
Time required by a binary search to search a list on n records is O(logn)

Best case for binary search happen if the search key is found in the middle array O(1).

Worse case for binary search happens if the search key is not in the list or the searching size equal to 1. The searching time for worst case is O(log2n).