DATA STRUCTURE AND ALGORITHMS

UNIT-1

Data Structure:

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e., Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Basic Terminology:

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned:-

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, 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 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

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

Advantages of Data Structures:

Efficiency:

Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

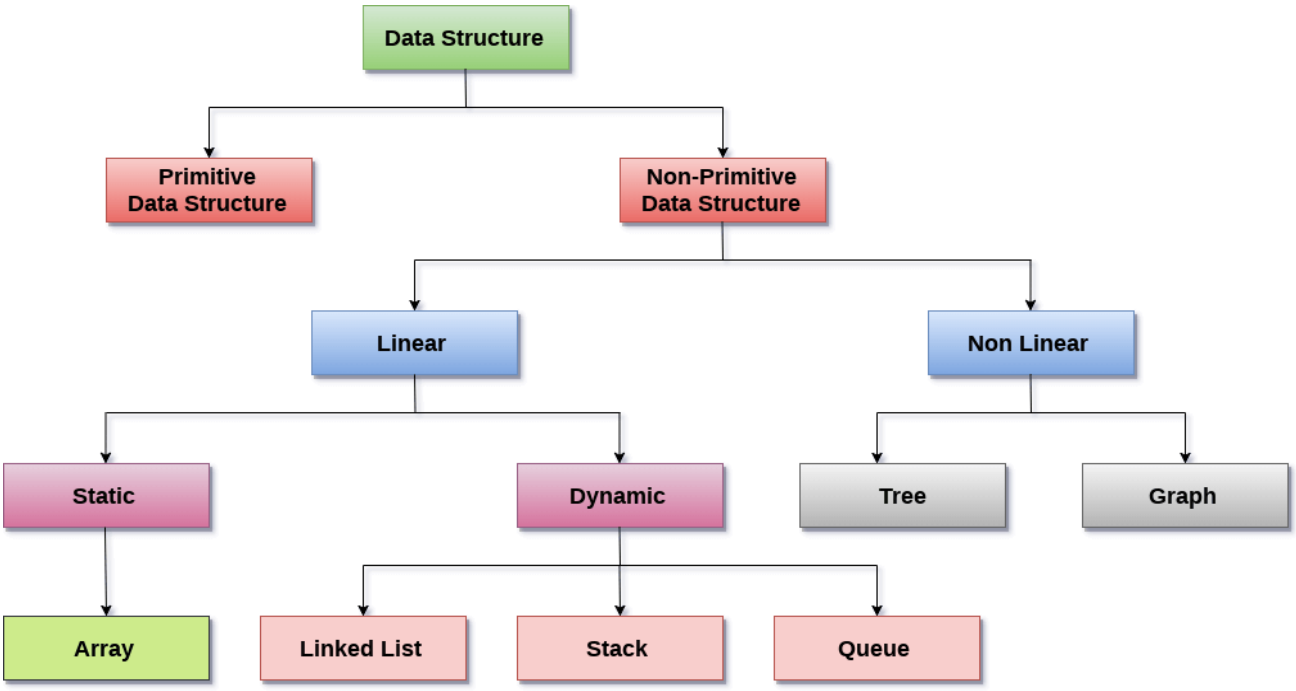
Reusability:

Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction:

Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Types of Data Structure:



Primitive data structure:

Primitive data structure is a data structure that can hold a single value in a specific location whereas the non-linear data structure can hold multiple values either in a contiguous location or random locations

The examples of primitive data structure are float, character, integer and pointer. The value to the primitive data structure is provided by the programmer. The following are the four primitive data structures:

- **Integer:** The integer data type contains the numeric values. It contains the whole numbers that can be either negative or positive. When the range of integer data type is not large enough then in that case, we can use long.
- **Float:** The float is a data type that can hold decimal values. When the precision of decimal value increases then the Double data type is used.
- **Boolean:** It is a data type that can hold either a True or a False value. It is mainly used for checking the condition.
- **Character:** It is a data type that can hold a single character value both uppercase and lowercase such as 'A' or 'a'.

Non-primitive data structure:

Non-primitive data structure is a type of data structure that can store the data of more than one type.

Types of non-primitive Data Structure:

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

1. Linear Data Structure:

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both ends therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non-Linear Data Structures:

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non-Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called leaf node while the topmost node is called root node. Each node contains pointers to point adjacent nodes.

Graphs: Graphs can be defined as the pictorial representation of the set of elements connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Primitive data structure	Non-primitive data structure
Primitive data structure is a kind of data structure that stores the data of only one type.	Non-primitive data structure is a type of data structure that can store the data of more than one type.
Examples of primitive data structure are integer, character, float.	Examples of non-primitive data structure are Array, Linked list, stack.
Primitive data structure will contain some value, i.e., it cannot be NULL.	Non-primitive data structure can consist of a NULL value.
The size depends on the type of the data structure.	In case of non-primitive data structure, size is not fixed.
It starts with a lowercase character.	It starts with an uppercase character.
Primitive data structure can be used to call the methods.	Non-primitive data structure cannot be used to call the methods.

Operations on data structure:

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.

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.

If we try to delete an element from an empty data structure then underflow occurs.

Searching: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

Merging: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

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.

The formal definition of an algorithm is that it contains 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 solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

Characteristics of an Algorithm:

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step-by-step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Algorithm Analysis:

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Importance of Analysis of Algorithms:

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Algorithm Analysis:

1. Best case
2. Worst case
3. Average case

1. Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search () function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be O(n).

2. Best Case Analysis (Very Rarely used)

In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So, time complexity in the best case would be Ω (1)

3. Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So, we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

Popular Notations in Complexity Analysis of Algorithms:

1. Big-O Notation

We define an algorithm's worst-case time complexity by using the Big-O notation, which determines the set of functions grows slower than or at the same rate as the expression. Furthermore, it explains the maximum amount of time an algorithm requires to consider all input values.

2. Omega Notation

It defines the best case of an algorithm's time complexity; the Omega notation defines whether the set of functions will grow faster or at the same rate as the expression. Furthermore, it explains the minimum amount of time an algorithm requires to consider all input values.

3. Theta Notation

It defines the average case of an algorithm's time complexity, the Theta notation defines when the set of functions lies in both O(expression) and Omega(expression), then Theta notation is used. This is how we define a time complexity average case for an algorithm.

Complexities of an Algorithm:

The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n). The complexity of an algorithm can be divided into two types. The time complexity and the space complexity.

Time Complexity of an Algorithm

The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm. This calculation is totally independent of implementation and programming language.

Space Complexity of an Algorithm

Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm. The memory space is generally considered as the primary memory.

Pseudocode:

A Pseudocode is defined as a step-by-step description of an algorithm. Pseudocode does not use any programming language in its representation instead it uses the simple English language text as it is intended for human understanding rather than machine reading. Pseudocode is the intermediate state between an idea and its implementation(code) in a high-level language.

How to write Pseudocode:

- Organize the sequence of tasks and write the pseudocode accordingly.
- At first, establishes the main goal or the aim.
- Use standard programming structures such as if-else, for, while, and cases the way we use them in programming. Indent the statements if-else, for, while loops as they are indented in a program, it helps to comprehend the decision control and execution mechanism. It also improves readability to a great extent.
- Use appropriate naming conventions. The human tendency follows the approach of following what we see. If a programmer goes through a pseudo code, his approach will be the same as per that, so the naming must be simple and distinct.
- Reserved commands or keywords must be represented in capital letters.
- Check whether all the sections of a pseudo code are complete, finite, and clear to understand and comprehend. Also, explain everything that is going to happen in the actual code.
- Don’t write the pseudocode in a programming language. It is necessary that the pseudocode is simple and easy to understand even for a layman or client, minimizing the use of technical terms.

Difference between Algorithm and Pseudocode:

Algorithm	Pseudocode

An Algorithm is used to provide a solution to a particular problem in form of a well-defined step-based form.	A Pseudocode is a step-by-step description of an algorithm in code-like structure using plain English text.
An algorithm only uses simple English words	Pseudocode also uses reserved keywords like if-else, for, while, etc.
These are a sequence of steps of a solution to a problem	These are fake codes as the word pseudo means fake, using code like structure and plain English text
There are no rules to writing algorithms	There are certain rules for writing pseudocode
Algorithms can be considered pseudocode	Pseudocode cannot be considered an algorithm
It is difficult to understand and interpret	It is easy to understand and interpret

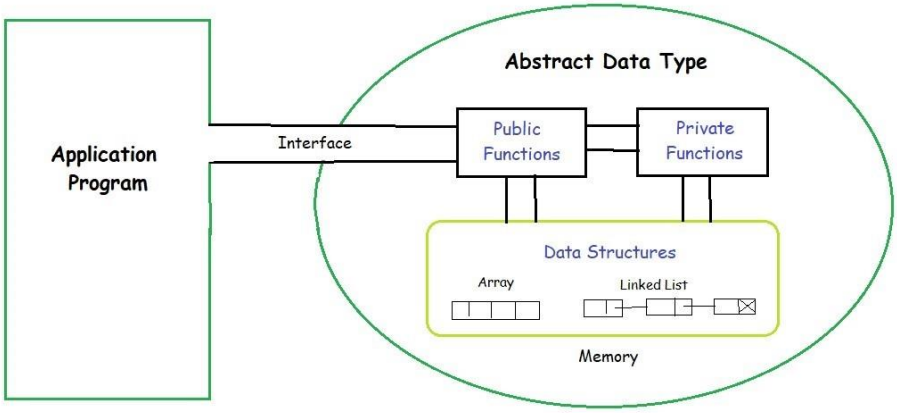
Difference between Flowchart and Pseudocode:

Flowchart	Pseudocode
A Flowchart is pictorial representation of flow of an algorithm.	A Pseudocode is a step-by-step description of an algorithm in code like structure using plain English text.
A Flowchart uses standard symbols for input, output decisions and start stop statements. Only uses different shapes like box, circle and arrow.	Pseudocode uses reserved keywords like if-else, for, while, etc.
This is a way of visually representing data, these are nothing but the graphical representation of the algorithm for a better understanding of the code	These are fake codes as the word pseudo means fake, using code like structure but plain English text instead of programming language
Flowcharts are good for documentation	Pseudocode is better suited for the purpose of understanding

Abstract Data Types:

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.



Features of ADT:

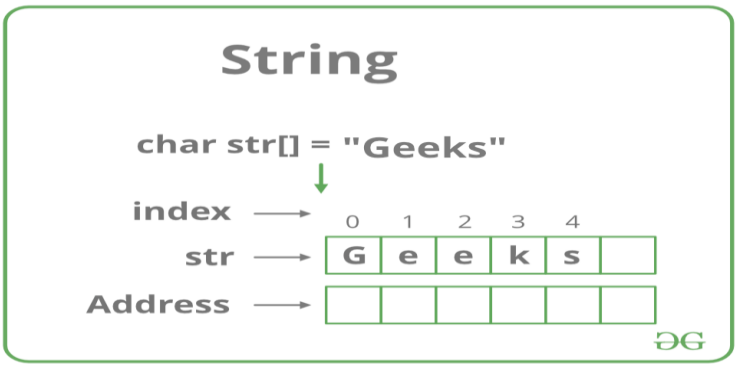
- **Abstraction:** The user does not need to know the implementation of the data structure.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.

String Data Structure:

String:

Strings are defined as an array of characters. The difference between a character array and a string is the string is terminated with a special character '\0'.

In C, a string can be referred to either using a character pointer or as a character array. When strings are declared as character arrays, they are stored like other types of arrays in C.



Advantages of String:

- String provides us a string library to create string objects which will allow strings to be dynamically allocated and also boundary issues are handled inside class library.

- String provides us various inbuilt functions under string library such as sort(), substr(i, j), compare (), push_back() and many more.
- In C language strings can have compile-time allocation and determination of size. This makes them more efficient, faster run-time at the time of using them.
- In C++ we do not need to predefine the size of a string.
- String helps as a base for many data structures such as tries, suffix trees, suffix arrays, ternary search trees, and much more.
- Strings provide us very helpful string algorithms for solving very complex problems with less time complexity.

Disadvantages of String:

- In JAVA strings are immutable they cannot be modified or changed
- Strings are generally slow in performing operations like input, output.
- In JAVA you cannot extend string class which means overriding methods in string class is not possible.
- C strings are fixed in size and are not dynamic.

Operations on String:

String provides users with various operations. Some of the important ones are:

size(): This function is used to find the length of the string.

substr(): This is used to find a substring of length a particular length starting from a particular index.

+: This operator is used to concatenate two strings.

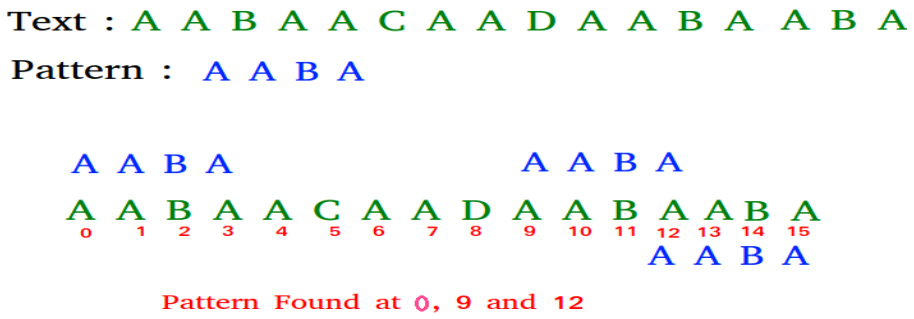
s1.compare(s2): This is used to compare two strings s1 and s2 to find which is lexicographically greater and which one is smaller.

reverse(): This function is used to reverse a given string.

sort(): This function is used to sort the string in lexicographic order.

Pattern Searching/Matching algorithms:

The Pattern Searching algorithms are sometimes also referred to as String Searching Algorithms and are considered as a part of the String algorithms. These algorithms are useful in the case of searching a string within another string.



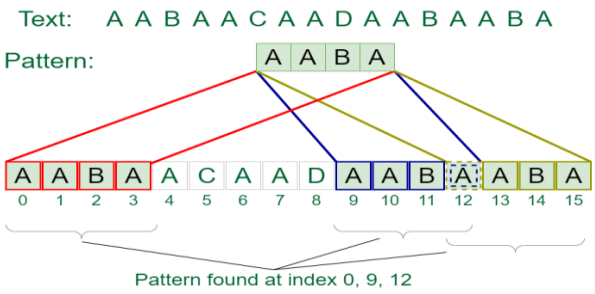
1. Naive algorithm for Pattern Searching

Given a text of length **N** txt[0..N-1] and a pattern of length M pat[0..M-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that N > M.

Examples:

Input: *txt[] = "THIS IS A TEST TEXT", pat[] = "TEST"*
Output: Pattern found at index 10

Input: *txt[] = "AABAACAADAABAABA", pat[] = "AABA"*
Output: Pattern found at index 0, Pattern found at index 9, Pattern found at index 12



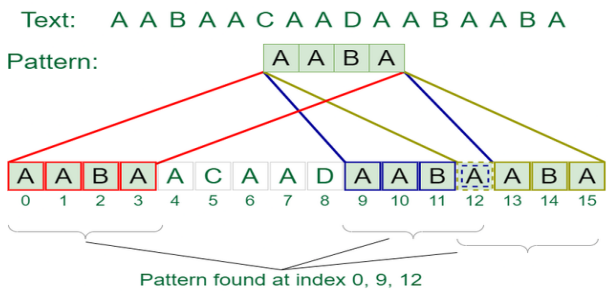
2. KMP Algorithm for Pattern Searching

Given a text txt[0..N-1] and a pattern pat[0..M-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that N > M.

Examples:

Input: *txt[] = "THIS IS A TEST TEXT", pat[] = "TEST"*
Output: Pattern found at index 10

Input: *txt[] = "AABAACAADAABAABA"*
pat[] = "AABA"
Output: Pattern found at index 0, Pattern found at index 9, Pattern found at index 12



Array:

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

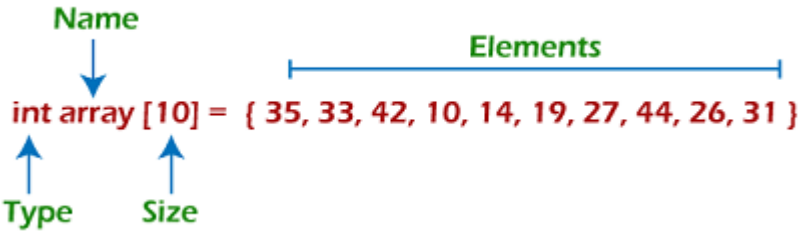
Properties of array:

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

We can represent an array in various ways in different programming languages. As an illustration, let's see the declaration of array in C language -



As per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

Basic operations:

Now, let's discuss the basic operations supported in the array -

- Traversal - This operation is used to print the elements of the array.
- Insertion - It is used to add an element at a particular index.
- Deletion - It is used to delete an element from a particular index.
- Search - It is used to search an element using the given index or by the value.
- Update - It updates an element at a particular index.

Insertion operation

This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array. Now, let's see the implementation of inserting an element into the array.

```
#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 operation:

As the name implies, this operation removes an element from the array and then reorganizes all of the array elements.

```
#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,
```

Searching in Arrays:

One of the basic operations to be performed on an array is searching. Searching an array means to find a particular element in the array. The search can be used to return the position of the element or check if it exists in the array.

Linear Search

The simplest search to be done on an array is the linear search. This search starts from one end of the array and keeps iterating until the element is found, or there are no more elements left (which means that the element does not exist).

There are no prerequisites for this search to work on an array. It can be used reliably in any situation.

Best use case

This search is best used when the list of elements is unsorted and the search is to be performed only once. It is also preferred for list to be small, as the time taken grows with the size of the data.

Time Complexity

- Average: O(n)
- Best: O(1)
- Worst: O(n)

The average time complexity is O(n) as the element to be found may be present at the end of the list or may not be present at all. Linear search has to visit all the elements until the needed element is found. Algorithmically, this comes out to be O(n).

The best and worst case of a search algorithm will be O(1) and O(n) respectively, as the element to be searched could always be found on the first iteration or the last iteration.

Binary Search

The linear search approach has one disadvantage. Finding elements in a large array will be time consuming. As the array grows, the time will increase linearly. A binary search can be used as a solution to this problem in some cases.

The principle of binary search is how we find a page in book. We open the book at a random page in the middle and based on that page we narrow our search to the left or right of the book. Indeed, this only is possible if the page numbers are in order.

Best use case

This search is best used when the list of elements is already sorted (not always feasible, especially when new elements are frequently being inserted). The list to be searched can be very large without much decrease in searching time, due to the logarithmic time complexity of the algorithm.

Time Complexity

- Average: O(log n)
- Best: O(1)
- Worst: O(n)

The average time complexity of this algorithm of searching is O(log n) as the number of elements to search halves during each iteration.

The best and worst case of a search algorithm will be O(1) and O(n) respectively, as the element to be searched could always be found on the first iteration or the last iteration.

Sorting in Arrays

Sorting an array means to arrange the elements in the array in a certain order. Various algorithms have been designed that sort the array using different methods. Some of these sorts are more useful than the others in certain situations.

Terminologies

Internal/External Sorting

Internal sorting means that all the data that is to be sorted is stored in memory while sorting is in progress.

External sorting means that the data is stored outside memory (like on disk) and only loaded into memory in small chunks. External sorting is usually applied in cases when data can’t fit into memory entirely, effectively allowing to sort data that does not fit in the memory.

Stability of Sort

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in the sorted output as they appear in the unsorted input.

A sorting algorithm is said to be unstable if there are two or more objects with equal keys which don't appear in same order before and after sorting.

Bubble Sort:

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. The pass through the list is repeated until the list is sorted.

This is an inefficient sort as it has to loop through all the elements multiple times. It takes O(n^2) time to completely sort the array.

Properties:

- Average Time Complexity: O(n^2)
- Stability: Stable

Best use case

This is a very elementary sort which is easy to understand and implement. It is not recommended in actual production environments. No external memory is required to sort as it is an in-place sort.

Multidimensional Arrays:

Multi-dimensional arrays are such type of arrays which stores another array at each index instead of single element. In other words, define multi-dimensional arrays as array of arrays. As the name suggests, every element in this array can be an array and they can also hold other sub-arrays within. Arrays or sub-arrays in multidimensional arrays can be accessed using multiple dimensions.

Two-dimensional array:

It is the simplest form of a multidimensional array. It can be created using nested array. These types of arrays can be used to store any type of elements, but the index is always a number. By default, the index starts with zero.

Syntax:

```
array (
   array (elements...),
   array (elements...),
```

```
    ...
)
```

Two-dimensional associative array:

Al associative array is similar to indexed array but instead of linear storage (indexed storage), every value can be assigned with a user-defined key of string type.

Three-Dimensional Array:

It is the form of multidimensional array. Initialization in Three-Dimensional array is same as that of Two-dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

Syntax:

```
array (
    array (
        array (elements...),
        array (elements...),
        ...
    ),
    array (
        array (elements...),
        array (elements...),
        ...
    ),
    ...
)
```


Pointer:

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

- Traversing String
- Lookup Tables
- Control Tables
- Tree Structures

Pointer Details

Pointer arithmetic: There are four arithmetic operators that can be used in pointers: ++, --, +, -

Array of pointers: You can define arrays to hold a number of pointers.

Pointer to pointer: C allows you to have pointer on a pointer and so on.

Passing pointers to functions in C: Passing an argument by reference or by address enable the past argument to be changed in the calling function by the called function.

Return pointer from functions in C: C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.