Unit 1

Data Structure

Definition, Types & Operations
Algorithm-basic concepts

Data type

A data type is an attribute of data which tells the compiler (or interpreter) how the programmer intends to use the data.

- Primitive: basic building block (int, float, char, double)
- Derived: any data type (struct, array, string etc.) composed of primitives or composite types.

Abstract data type

An abstract data type (**ADT**) is a mathematical model for data types. An ADT may be defined as a "class of objects whose logical behavior is defined by a set of values and a set of operations". ADTs are a theoretical concept, in computer science, used in the design and analysis of algorithms, data structures, and software systems, and do not correspond to specific features of computer languages.

Data Structure

In many ways we can define data structures:

- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising 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.

- A data structure is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

- Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.
    - Interface − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
    - Implementation − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

- Correctness − Data structure implementation should implement its interface correctly.
- Time Complexity − Running time or the execution time of operations of data structure must be as small as possible.
- Space Complexity − Memory usage of a data structure operation should be as little as possible.

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 particular 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.

Need for Data Structure

As applications are getting complex and data rich, there may be problems such as,

- Data Search − As data grows, search will become time consuming.
- Processor speed − Processor speed although being very high, falls limited if the data grows to billion records.
- Multiple requests − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

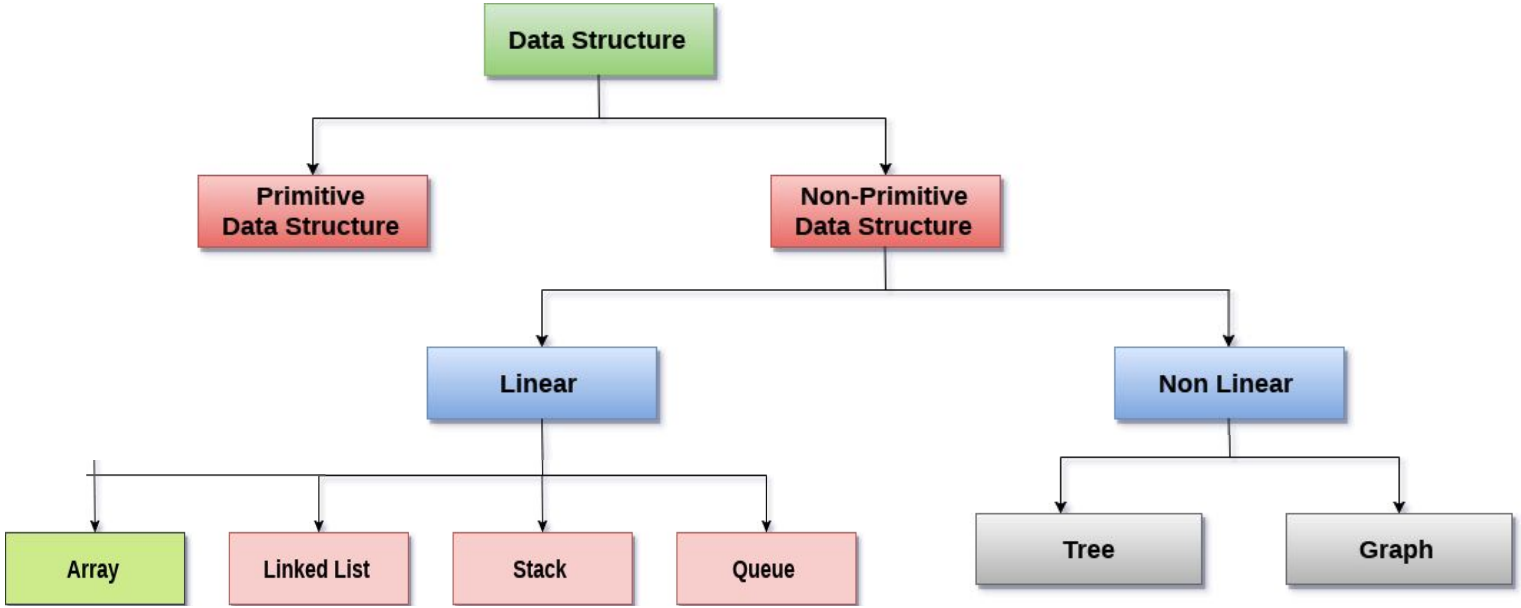
To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases:

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

- Worst Case − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is $f(n)$ then this operation will not take more than $f(n)$ time where $f(n)$ represents function of n.
- Average Case − This is the scenario depicting the average execution time of an operation of a data structure.
- Best Case − This is the scenario depicting the least possible execution time of an operation of a data structure.

Data Structure Classification



Data structures can be classified as Primitive and Non-primitive data structures.

Primitive Data Structures:

These are the structures which are supported at the machine level, they can be used to make non-primitive data structures. These are integral and are pure in form. They have predefined behavior and specifications.

Examples: Integer, float, character, pointers.

The pointers, however don't hold a data value, instead, they hold memory addresses of the data values. These are also called the reference data types.

Non-primitive Data Structures:

The non-primitive data structures cannot be performed without the primitive data structures. Although, they too are provided by the system itself yet they are derived data structures and cannot be formed without using the primitive data structures.

Examples: Array, record, file, list, stack, queue, tree, graph…

Record: A record is a collection of related data items. Eg: A student record contains name, roll number, class, mark etc.

File: A file is a collection of records. The file data structure is primarily used for managing large amounts of data which is not in the primary storage of the system. The files help us to process, manage, access and retrieve or basically work with such data, easily.

The Non-primitive data structures are further divided into the following categories:

- **Linear and Non-linear data structures**
- **Homogeneous and Non-homogeneous (Heterogeneous) data structures**
- **Static and Dynamic data structures**
- **Contiguous and Non-contiguous data structures**

Linear and Non-linear data structures

Linear Data Structures: 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. The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

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**. Also known as Last-In-First-Out (LIFO) list. 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 end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

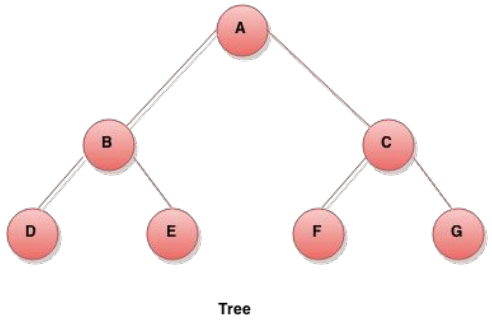
Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

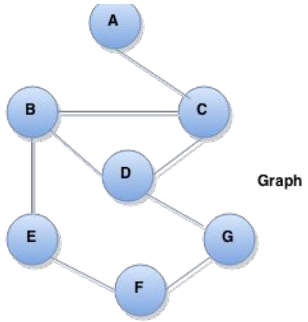
Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf nodes** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) 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 can not have the one.



Tree



Graph

Static and Dynamic data structures

Static data structures:

A static data structure is an organization or collection of data in memory that is fixed in size. This results in the maximum size needing to be known in advance, as memory cannot be reallocated at a later point. A key advantage of static data structures is that with memory allocation fixed, no control or oversight is needed to prevent potential overflow or underflow issues when adding new items or removing existing ones. This makes static data structures easier to program but at the expense of potential efficiency in terms of memory consumption.

Example: Arrays

Dynamic data structures

A dynamic data structure refers to an organization or collection of data in memory that has the flexibility to grow or shrink in size, enabling a programmer to control exactly how much memory is utilized. Dynamic data structures change in size by having unused memory allocated or de-allocated from the heap as needed. Dynamic data structure has the flexibility to consume additional memory if needed or free up memory when possible for improved efficiency.

Example: Linked list

Homogeneous and Heterogeneous data structures

Homogeneous data structures:

Homogeneous data structures are those data structures that contain only similar type of data.

Example: Arrays

Heterogeneous data structures

Heterogeneous Data Structures are those data structures that contains a variety or dissimilar type of data.

Example: Records, Structure, Union

Contiguous and Non-contiguous data structures

Contiguous data structures:

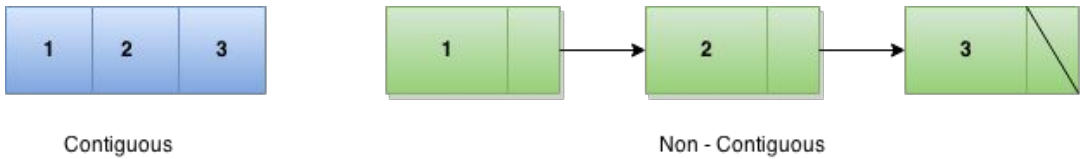
In contiguous data structures, data items are kept together in memory (either RAM or in a file) at continuous locations.

Example: Arrays

Non-contiguous data structures:

In non-contiguous data structures, data items are scattered in memory, but all the data items are linked to each other in some way.

Example: Linked list (the nodes of the list are linked together using pointers stored in each node).



Operations on data structure

1) **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.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

3) **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.

4) **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.

5) **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.

6) **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

...etc.…

What is an Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Every Algorithm must satisfy the following properties:

1. **Input**- There should be 0 or more inputs supplied externally to the algorithm.
2. **Output**- There should be atleast 1 output obtained.
3. **Definiteness**- Every step of the algorithm should be clear and well defined.
4. **Finiteness**- The algorithm should have finite number of steps.
5. **Correctness**- Every step of the algorithm must generate a correct output.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

Space Complexity

Space complexity is the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space:** It is the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** It is the space required to store all the constants and variables(including temporary variables) value.
- **Environment Stack Space:** It is the space required to store the environment information needed to resume the suspended function.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack Space**.

Time Complexity

Time Complexity is a way to represent the amount of processor time required by the program to run till its completion. It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible. Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.

There are two methods for calculating time complexity. **Step count method** and **operation count method**.

Step Count Method

The step count method is one of the method to analyze the algorithm. In this method, we count number of times one instruction is executing. From that we will try to find the complexity of the algorithm.

Operation Count Method

We can estimate the time complexity of an algorithm by choosing one of different operations (that is, the major operation). Operation count method checks, how many times the major operation executes. The success of this method depends on the ability to identify the operation that contributes most of the time complexity.