UNIT 1 :

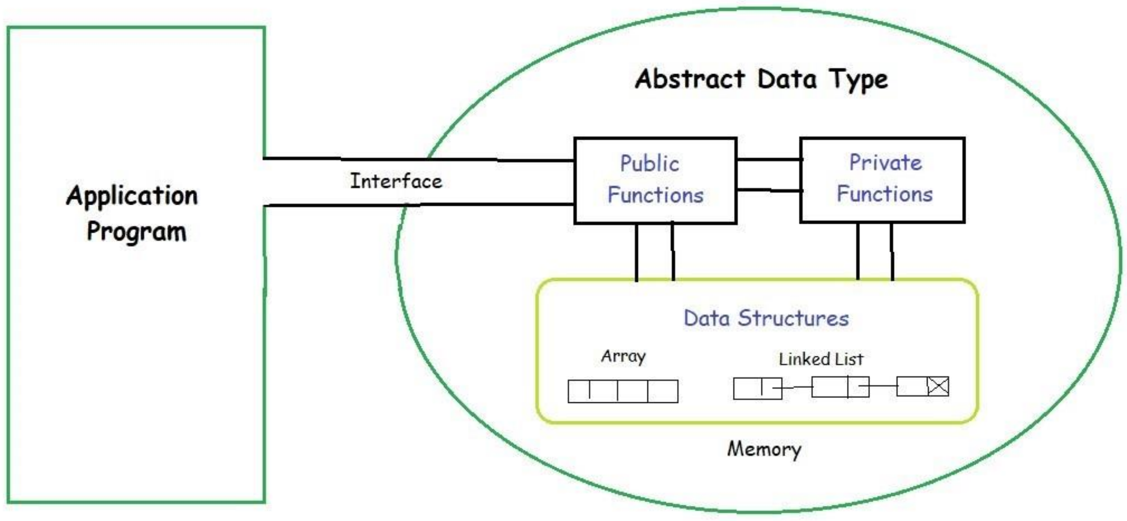
Basic concepts of Data Structures:

Abstract data types:

In this article, we will learn about ADT but before understanding what ADT is let us consider different in-built data types that are provided to us. Data types such as int, float, double, long, etc. are considered to be in-built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a situation when we need operations for our user-defined data type which have to be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as Abstract Data Type (ADT).

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.



The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.
So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

Features of ADT:

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.

- **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner.

Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development. Here are some of the main advantages and disadvantages of using ADTs:

What are Fundamental Datatypes?

A **Fundamental Datatype** is one which is a concrete form of data type, and it is introduced by the programming language itself. Therefore, a fundamental datatype has its own fundamental characteristics and properties defined in the language. It also has some fundamental methods to perform operations over the data.

In case of fundamental datatypes, the only concern is the type and nature of the data. There is no issue of time complexity because we deal with the concrete implementation of the programming language. Some common fundamental datatypes include **int, char, float, void,** etc.

What are Derived Datatypes?

Derived Datatypes are composed of fundamental datatypes; they are derived from the fundamental data types. Therefore, they have some additional characteristics and properties other than that of fundamental data types.

There is an issue of time complexity in the case of derived data types because they deal with manipulation and execution of logic over data that it stores. Derived data types are defined by the user because the programming language does not have built-in definition of the derived data types.

Programmers can modify or redefine the derived datatypes. Some common examples of derived datatypes include **arrays, structures, pointers,** etc.

Difference between Fundamental Data Types and Derived Data Types

The following table highlights the important differences between fundamental datatypes and derived datatypes −

S.No.	Fundamental Datatypes	Derived Datatypes
1.	Fundamental datatypes are also known as primitive datatypes.	Derived datatypes are composed of fundamental datatypes.

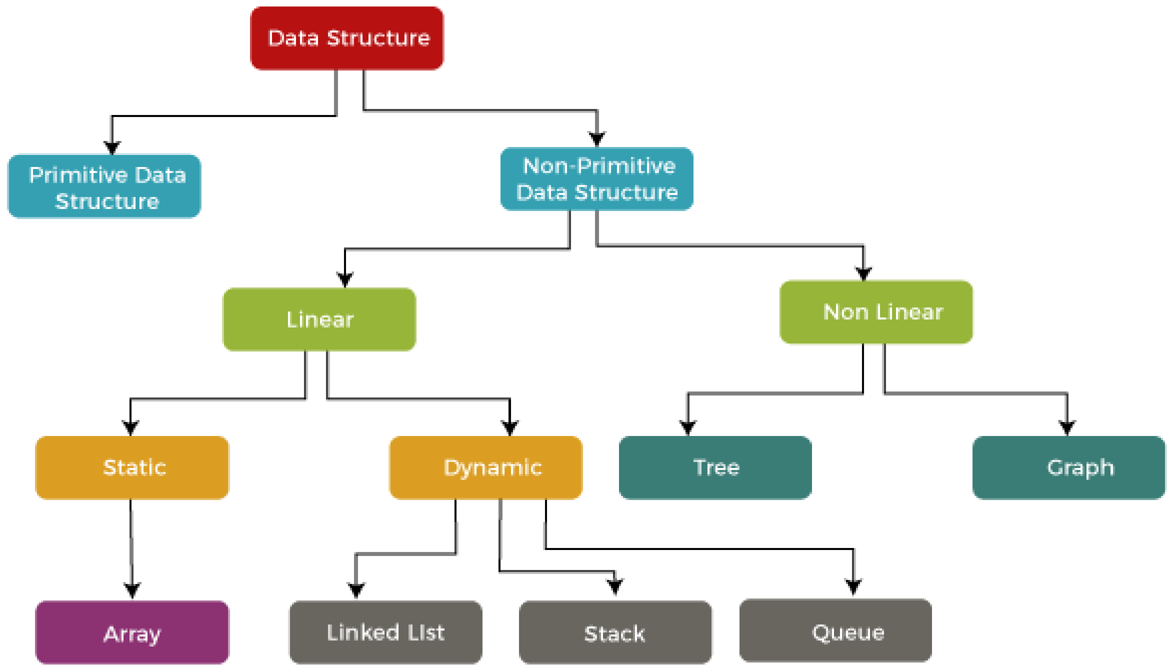
2.	Some fundamental datatypes include **int, char, float, void,** etc.	Derived datatypes include arrays, structures, pointers, etc.
3.	Integer or Character datatypes are classified as **int, char, signed int, signed char, unsigned int, unsigned char.**	Pointers are used to store address of some other variables.
4.	Integers are used to store integer type data, not the floating point number.	Arrays are used to store homogeneous data.
5.	Floats are used to store decimal numbers. The variations **include float, double, long double.**	Structures are group of some primitive datatypes like int, **float, double,** etc.
6.	**Voids is** used where no return values are specified.	Unions are like structures, but all the members of a union share the same memory location.

Conclusion

To conclude, fundamental datatypes are those which have their own fundamental characteristics and properties along with some fundamental methods to perform operations over them, whereas derived datatypes are those which are derived from fundamental data types and have some additional or modified properties other than that of the fundamental data types.

Primitive data structure

Primitive data structure is a fundamental type of data structure that stores the data of only one type whereas the non-primitive data structure is a type of data structure which is a user-defined that stores the data of different types in a single entity.



In the above image, we can observe the classification of the data structure. The data structure is classified into two types, i.e., primitive and non-primitive data structure. In the case of primitive data structure, it contains fundamental data types such as integer, float, character, pointer, and these fundamental data types can hold a single type of value. For example, integer variable can hold integer type of value, float variable can hold floating type of value, character variable can hold character type of value whereas the pointer variable can hold pointer type of value.

Design and analysis of algorithm:

The Need for Analysis

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.
- **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

Rate of Growth

Rate of growth is defined as the rate at which the running time of the algorithm is increased when the input size is increased.

The growth rate could be categorized into two types: linear and exponential. If the algorithm is increased in a linear way with an increasing in input size, it is **linear growth rate**. And if the running time of the algorithm is increased exponentially with the increase in input size, it is **exponential growth rate**.

Proving Correctness of an Algorithm

Once an algorithm is designed to solve a problem, it becomes very important that the algorithm always returns the desired output for every input given. So, there is a need to prove the correctness of an algorithm designed. This can be done using various methods −

Proof by Counterexample

Identify a case for which the algorithm might not be true and apply. If the counterexample works for the algorithm, then the correctness is proved. Otherwise, another algorithm that solves this counterexample must be designed.

Proof by Induction

Using mathematical induction, we can prove an algorithm is correct for all the inputs by proving it is correct for a base case input, say 1, and assume it is correct for another input k, and then prove it is true for k+1.

Proof by Loop Invariant

Find a loop invariant k, prove that the base case holds true for the loop invariant in the algorithm. Then apply mathematical induction to prove the rest of algorithm true.

In System Design, there are two types of approaches followed namely, the **Bottom-Up Model** and the **Top-Down Model**.

- The bottom-up model is one in which the different parts of a system are designed and developed and then all these parts are connected together as a single unit.
- On the other hand, the top-down model is one in which the whole system is decomposed into smaller sub-components, then each of these parts are designed and developed till the completed system is designed.

Read this article to find out more about the bottom-up model and the top-down model of system design and how they are different from each other.

Top-down and bottom up approaches to algorithm Design

What is Bottom-Up Model?

Bottom-Up Model is a system design approach where the parts of a system are defined in details. Once these parts are designed and developed, then these parts or components are linked together to prepare a bigger component. This approach is repeated until the complete system is built.

The advantage of Bottom-Up Model is in making decisions at very low level and to decide the re-usability of components.

What is Top-Down Model?

Top-Down Model is a system design approach where the design starts from the system as a whole. The complete system is then divided into smaller sub-applications with more details.

Each part again goes through the top-down approach till the complete system is designed with all the minute details. TopDown approach is also termed as breaking a bigger problem into smaller problems and solving them individually in recursive manner.

Conclusion

The most significant difference between the two types of models is that the bottom-up model is based on the composition approach, while the top-down model is based on the decomposition approach.

Another important difference between the two is that the top-down model is mainly used in structural programming like C programming, whereas the bottom-up approach is followed in objectoriented programming like C++, Java, etc.

Frequency Count.

Frequency count specifies the number of times a statement is to be executed, so how many times the statement within our algorithm is executed

Algorithm Analysis

Analysis of efficiency of an algorithm can be performed at two different stages, before implementation and after implementation, as

A priori analysis − This is defined as theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. speed of processor, are constant and have no effect on implementation.

A posterior analysis − This is defined as empirical analysis of an algorithm. The chosen algorithm is implemented using programming language. Next the chosen algorithm is executed on target computer machine. In this analysis, actual statistics like running time and space needed are collected.

Algorithm analysis is dealt with the execution or running time of various operations involved. Running time of an operation can be defined as number of computer instructions executed per operation.

Algorithm Complexity

Suppose X is treated as an algorithm and N is treated as the size of input data, the time and space implemented by the Algorithm X are the two main factors which determine the efficiency of X.

Time Factor − The time is calculated or measured by counting the number of key operations such as comparisons in sorting algorithm.

Space Factor − The space is calculated or measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(N) provides the running time and / or storage space needed by the algorithm with respect of N as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

Space needed by an algorithm is equal to the sum of the following two components

A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

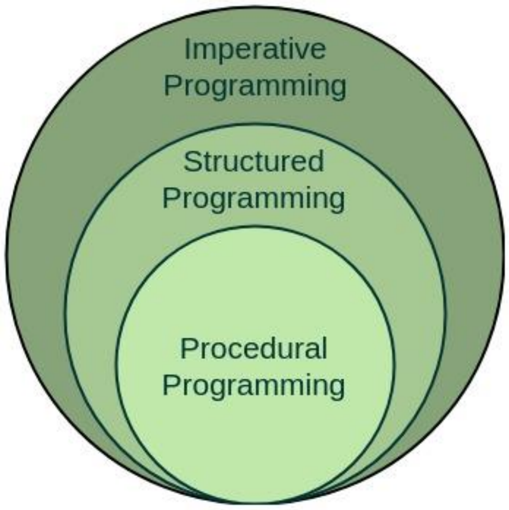
Space complexity S(p) of any algorithm p is S(p) = A + Sp(I) Where A is treated as the fixed part and S(I) is treated as the variable part of the algorithm which depends on instance characteristic I. Following is a simple example that tries to explain the concept

Structured Programming Approach with Advantages and Disadvantages

Structured Programming Approach, as the word suggests, can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are:

- C
- C++
- Java
- C#

..etc



On the contrary, in the Assembly languages like Microprocessor 8085, etc, the statements do not get executed in a structured manner. It allows jump statements like GOTO. So the program flow might be random.

The structured program mainly consists of three types of elements:

- Selection Statements
- Sequence Statements
- Iteration Statements

The structured program consists of well structured and separated modules. But the entry and exit in a Structured program is a single-time event. It means that the program uses single-entry and single-exit elements. Therefore a structured program is well maintained, neat and clean program. This is the reason why the Structured Programming Approach is well accepted in the programming world.

Advantages of Structured Programming Approach:

1. Easier to read and understand
2. User Friendly
3. Easier to Maintain
4. Mainly problem based instead of being machine based
5. Development is easier as it requires less effort and time
6. Easier to Debug
7. Machine-Independent, mostly.

Disadvantages of Structured Programming Approach:

1. Since it is Machine-Independent, So it takes time to convert into machine code.
2. The converted machine code is not the same as for assembly language.
3. The program depends upon changeable factors like data-types. Therefore it needs to be updated with the need on the go.
4. Usually the development in this approach takes longer time as it is language-dependent. Whereas in the case of assembly language, the development takes lesser time as it is fixed for the machine.