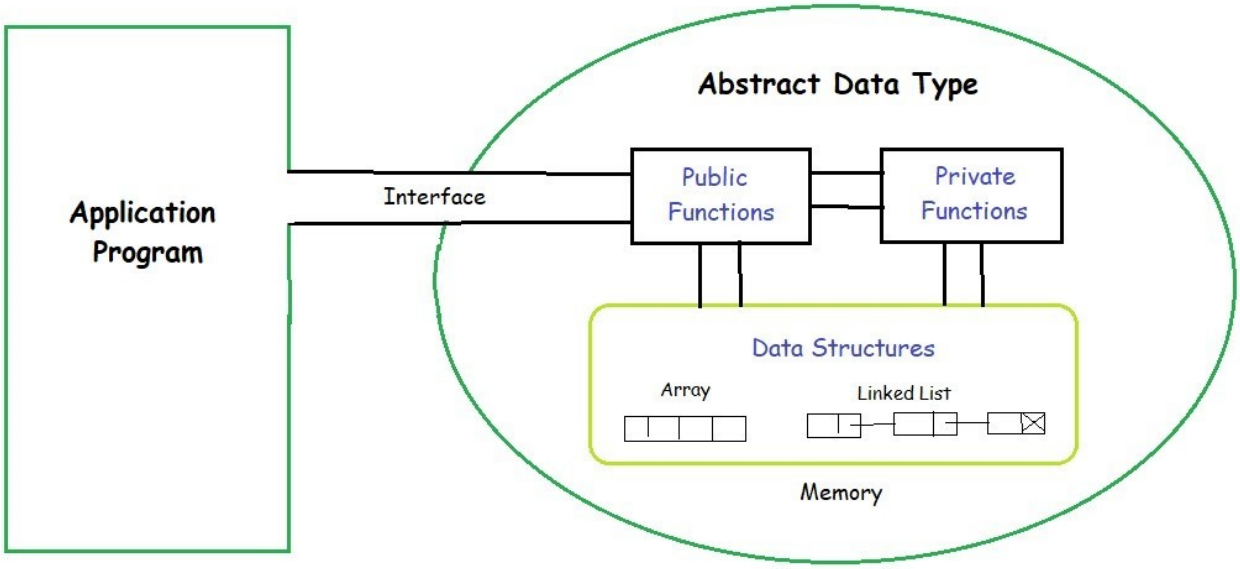
Dr. B.S.R.V. Prasad
Department of Mathematics

Abstract Data Type (ADT)

Data types such as int, float, double, and long are built-in types that allow us to perform basic operations like addition, subtraction, division, and multiplication. However, there are scenarios where we need custom operations for different data types. These operations are defined based on specific requirements and are tailored as needed. To address such needs, we can create data structures and their operations, known as Abstract Data Types (ADTs).

An **Abstract Data Type (ADT)** is a conceptual model that defines a set of operations and behaviours for a data type without specifying how these operations are implemented or how data is organised in memory. 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 organised in memory and what algorithms will be used to implement the operations. It is called "abstract" because it provides an implementation-independent view.

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



Features of ADT:

Abstract data types (ADTs) encapsulate 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 conceptualisation of the real world.
- **Robust:** The program is robust and can 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 their functionality.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorised 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.

Advantages:

- **Encapsulation**: ADTs provide a way to encapsulate data and operations into a single unit, making managing and modifying the data structure easier.
- **Abstraction:** ADTs allow users to work with data structures without knowing the implementation details, which can simplify programming and reduce errors.
- **Data Structure Independence:** ADTs can be implemented using different data structures, making adapting to changing needs and requirements easier.
- **Information Hiding:** ADTs can protect data integrity by controlling access and preventing unauthorised modifications.
- **Modularity:** ADTs can be combined with other ADTs to form more complex data structures, which can increase programming flexibility and modularity.

Disadvantages:

- **Overhead:** Implementing ADTs can add overhead regarding memory and processing, affecting performance.
- **Complexity:** ADTs can be complex to implement, especially for large and complex data structures.
- **Learning Curve:** Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- **Limited Flexibility:** Some ADTs may be limited in functionality or unsuitable for all types of data structures.
- **Cost:** Implementing ADTs may require additional resources and investment, which can increase the cost of development.