Data Structures

Dilip Kumar Maripuri
Computer Applications

Data Structures

Session : Data Types, Data Structures, Abstract Data Types

Dilip Kumar Maripuri
Computer Applications

- **Topics Outline**
  - Uses of Data, Information
  - Introduction to Data Structures
  - Introduction to Abstract Data Types

- What is the most important factor for any event/ task to take place?
  - Any task or job done involves the movement/ processing of data.

The use of Data!!

- **The Nature of Data**
  - Data are individual units of raw facts or figures without context.
  - For example, 42, blue, or 15 mph are all pieces of data.
  - On their own, these items lack meaning.
- **Data in Organizations**
  - In the context of an organization, data are the building blocks that, when aggregated, analyzed, and interpreted, become information.
  - This information is essential for decision-making processes.

- **The Data Resource Circle** : Represents the organization's comprehensive collection of raw data elements that individually lack meaning but together form the organization's core data resource.
- **Data Transformation into Information** : Raw data becomes valuable information when aggregated, classified, and interpreted, enabling better understanding and insights.
- **Decision-Making Process** : Information derived from data aids managers and decision-makers in achieving objectives, solving problems, and leveraging opportunities effectively.

- **Action and Reaction** : Decisions lead to actions, such as launching marketing campaigns based on sales information, which in turn generate more data.
- **Data Use Flexibility** : Data can be combined in different ways to yield multiple insights. For example, sales data can inform inventory restocking, trend analysis, and sales strategies.
- **Managing Data as an Asset** : Effective data management is crucial for ensuring integrity, availability, and generating valuable insights, making it a critical investment.

Data

- Data is Expensive !!!
- Managing data effectively is a critical investment for any organization.
- To ensure data integrity and availability for generating valuable information, several key processes must be meticulously executed.

- Data structures relate to the way data is organized, managed, and stored in a computer so that it can be accessed and modified efficiently.
- In programming, data structures are used to store and organize data so that various operations can be performed on this data in an effective way.

- **Need for Knowledge**
  - Programmers, whether they work on systems or applications software, must understand data structures to create efficient and effective programs.
- **Data Representation**
  - In a program, data can be stored temporarily in the program's memory or permanently in a file.
- **Relationships Among Data**
  - Data values are often interconnected, and these relationships can be leveraged by programs to perform complex tasks.

- **Organization of Data**
  - To utilize these relationships, data must be structured in an organized manner.
  - This organization of data enables programs to access and manipulate the data efficiently.
- **Definition of Data Structure**
  - A data structure is thus an organized collection of data, combined with a set of defined operations that can be performed on it.
  - This organization not only stores data but also facilitates its manipulation.

- **Rules for Data Access**
  - Programs must adhere to specific rules for accessing and processing structured data.
  - These rules ensure that data is handled in a consistent and predictable manner.

Data Structure = Organized Data + Allowed Operations

- It is important for the data to be arranged in an organized manner for the applications to make use of the data relationships.
- **Data Structure** studies the organization of data in computers, consisting of
  - the (abstract) data types (definition and representation),
  - relationship between elements of this type,
  - operations on data types.
- **Algorithms**
  - methods to operate on data structures,
  - tradeoff between efficiency and simplicity,
  - subtle interplay with data structure design.
- **Program = data structures + algorithm**

- **Primitive Data Types**
  - Primitive data types are the most basic forms of data representation that are supported natively by a programming language.
  - They typically include types like integers, floats, characters, and booleans.

Characteristics

  - **Built-in:** Defined by the language itself and not created by the programmer.
  - **Simple:** Represent single values and not a combination of values.
  - **Fixed Size:** Occupy a fixed amount of memory that is known at compile-time.

Non-Primitive Data Types

- Non-primitive data types, also known as composite or reference types, are more complex structures that are built using primitive types and other non-primitive types.
- They include arrays, lists, classes, and more.

Characteristics

- **Constructed:** Created using one or more primitive or non-primitive data types.
- **Aggregated:** Can represent a collection or a complex entity with multiple properties.
- **Variable Size:** Their size can change during runtime, particularly for dynamic data structures.

- **Simple Data Types to Scalar Types**
  - In programming languages like C, simple data types like integers and characters can be combined to create new scalar types, such as enumerated types.
  - An enumerated type is a user-defined data type that consists of integral constants and each of them is assigned a unique name.
  - A visual of an enumerated type listing color constants (RED, GREEN, BLUE).

- **Standard Data Structures as Building Blocks**
  - Arrays are one of the fundamental data structures used to store a collection of elements (usually of the same type).
  - They serve as building blocks for more complex structures by providing a way to access data via indices.

- **Linear vs. Non-Linear**
  - **Linear Data Structures**
    - Elements are arranged in a linear order where each element has a single predecessor and successor.
    - Arrays and linked lists are examples of linear structures.
  - **Non-Linear Data Structures**
    - Elements are not arranged sequentially and can have multiple relationships.
    - Trees and graphs are examples of non-linear structures.

- **Homogeneous vs. Non-Homogeneous**
  - **Homogeneous Data Structures**
    - All elements are of the same type, such as an array of integers or floats.
  - **Non-Homogeneous Data Structures**
    - Elements can be of different types. Structures like records (also known as structs in C) are common examples where different data types can be stored.

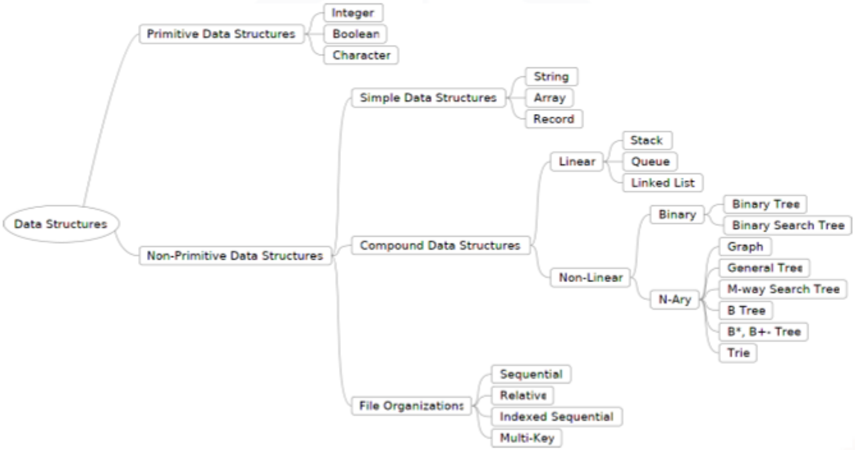
- **Static vs. Dynamic**

- **Static Data Structures**
  - The size and structure are defined at compile-time and do not change during execution.
  - Standard arrays in C are static because their size must be known at compile-time.

- **Dynamic Data Structures**
  - These can grow or shrink during execution, and their memory allocation is performed during runtime.
  - Linked lists and dynamic arrays are examples of dynamic structures.

Data Structures

Data Types



Abstract Data Types

- An Abstract Data Type (**ADT**) is a theoretical concept in computer science that defines a data structure solely by the operations that can be performed on it and the mathematical or logical properties of those operations.
- It provides a high-level, user-oriented description of what the data structure should do, rather than how it should do it.
- This means that an **ADT** specifies the interface that the data structure must adhere to, without any concern for the actual implementation.

Abstract View of Data

- ADTs provide a high-level, mathematical view of a data structure that focuses on what the data represents and what operations can be performed on it.
- The 'abstract' part means we are not concerned with how the data is stored or how the operations are carried out.

Data and Operations

- An ADT is defined by a set of data and a corresponding set of operations.
- For example, an ADT for a 'Stack' might include data that are elements of the stack and operations like 'push' (to add an item), 'pop' (to remove an item), and 'peek' (to view the top item without removing it).

- **Mathematical Construct**
  - The idea of an ADT is mathematical because it deals with the logic and rules that govern the data type.
  - It specifies what operations are allowed, what conditions apply to them, and what results are expected.

- **Value Collection**
  - The ADT specifies a collection of values.
  - These values could be of any type, such as integers, strings, or even other ADTs.
- **Operation Set**
  - It specifies a set of operations.
  - These operations define how we can interact with the data values.
  - They form the interface through which all interactions with the data are made.

- **Implementation Agnostic**
  - ADTs do not specify how the data and operations should be implemented.
  - This means an ADT can be implemented in multiple ways, using different data structures or programming languages.
- **Logical Properties**
  - In defining an ADT, the focus is on the logical properties and behaviors of the data type, not on performance or memory usage.

- **Specification for Implementers**
  - For someone implementing an ADT, it acts as a blueprint.
  - It tells them what the ADT must do, but not how to do it.
  - This allows for innovation and optimization in how the ADT is implemented.
- **Guideline for Programmers**
  - For programmers using an ADT, it acts as a contract.
  - It guarantees that certain operations will behave in a certain way, regardless of how the ADT is implemented.

- There are several ways of defining an **ADT**
  - **Mathematical Notion**
    - No focus on Time / Space Complexity.
    - Not Concerned with Implementation Details.
    - Acts as a guideline to the implementors.
  - **Semi-Formal**
    - Use of C Notions

- An **ADT** consists of two parts:
  - A value definition for the **ADT**
    - A definition clause
    - A condition clause
  - An operator definition.

- A Fraction is defined as $\frac{a}{b}$ where $b \neq 0$
- **Possible operations on fractions**
  - Creation of a fraction
  - Addition of 2 Fractions
  - Comparison of 2 Fractions
  - Multiplication of 2 Fractions

- **Abstract Data Type: Fraction**
  - This **ADT** represents a mathematical fraction, which is composed of two parts:
    - A numerator and a denominator.
  - **Data Elements**
    - **Numerator (Fraction_0)**: The top part of the fraction, representing how many parts of a whole are being considered.
    - **Denominator (Fraction_1)**: The bottom part of the fraction, representing the total number of equal parts the whole is divided into.
  - **Condition**
    - The denominator (Fraction1) must not be zero, as division by zero is undefined.

- **Value Definition**

abstract typedef <integer, integer> RATIONAL;

Condition RATIONAL[1] != 0;

- **Operator Definition**

abstract RATIONAL makerational (a, b)

int a, b;

precondition b != 0;

postcondition makerational[0] == a;

makerational[1] == b;

- **Operations on RATIONAL**

abstract RATIONAL add (a, b) /* written a + b */
RATIONAL a, b;
Postcondition add[1] == a[1] * b[1];
add[0] == a[0] * b[1] + b[0] * a[1];
abstract RATIONAL mult (a, b) /* written a * b */
RATIONAL a, b;
postcondition mult[0] == a[0] * b[0];
mult[1] == a[1] * b[1];
abstract equal (a, b) /* written a == b */
RATIONAL a, b;
postcondition equal == (a[0] * b[1] == b[0] * a[1]);

- **COMPLEX Data Type Definition and Operations** /* value definition */

  abstract typedef <number, number> COMPLEX;

  /* operator definition */

  abstract COMPLEX makecomplex (a, b)

  number a, b;

  postcondition makecomplex == a ± ib;

  abstract COMPLEX add (x, y)

  COMPLEX x, y;

  postcondition add[0] == x[0] + y[0];

  add[1] == x[1] + y[1];

  add == add[0] ± i.add[1];

Thank You

Dilip Kumar Maripuri
Associate Professor
Department of Computer Applications
dilip.maripuri@pes.edu
8073212026