DATA STRUCTURE

Chapter 1: Introduction to Data Structure

Engineering / Diploma — Study Notes

1.1 Data Representation

Data representation refers to the methods and formats used to store, organize, and encode data inside a computer's memory so that it can be processed efficiently by a program. Since computers only understand binary signals (0s and 1s), every piece of data — whether it is a number, a character, an image, or an instruction — must ultimately be represented in binary form. How data is represented directly affects the amount of memory it occupies, the speed of processing, and the range of operations that can be performed on it.

The way data is represented also determines which data structure is suitable for a given problem. For example, a set of related values of the same type is best represented using an array, whereas hierarchical relationships are better represented using a tree. Good data representation improves memory utilization and program efficiency.

Key points on data representation:

- All data is ultimately stored in binary form (bits and bytes) inside computer memory.
- Representation depends on the type of data: numeric (integer, float), character, boolean, or composite (arrays, records).
- Efficient representation reduces memory wastage and speeds up access and manipulation.
- The choice of representation influences which operations (search, sort, insert, delete) can be performed efficiently.
- Representation may be fixed-size (as in arrays) or variable-size (as in linked structures).

1.2 Abstract Data Types (ADT)

An Abstract Data Type (ADT) is a mathematical or logical model of a data type that is defined purely by the set of values it can hold and the operations that can be performed on it, without specifying how these values and operations are actually implemented in a programming language. In other words, an ADT tells us WHAT operations are possible, but hides HOW those operations are carried out internally.

The concept of ADT is central to the idea of data abstraction and encapsulation in programming. A user of an ADT only needs to know the interface (the operations available), not the internal implementation details such as the underlying data structure, memory allocation, or algorithm used. This separation allows programmers to change the internal implementation of an ADT without affecting the programs that use it.

Examples of common ADTs:

- Stack — supports push, pop, and peek operations following Last-In-First-Out (LIFO) order.
- Queue — supports enqueue and dequeue operations following First-In-First-Out (FIFO) order.

- List — supports insertion, deletion, and traversal of an ordered collection of elements.
- Set — supports union, intersection, and membership-checking operations on a collection of unique elements.
- Map/Dictionary — supports insertion, deletion, and lookup of key-value pairs.

Characteristics of an ADT:

- Defines data and operations together, but hides implementation details.
- Provides a well-defined interface for interaction with the data.
- Promotes modularity, reusability, and easier maintenance of code.
- Can be implemented using different underlying data structures (e.g., a Stack ADT can be implemented using an array or a linked list).

1.3 Data Structure and Structured Types

A data structure is a specific way of organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently. It defines the relationship between data elements and the operations that can be performed on them. A data structure is essentially the practical implementation of an ADT.

A structured type (also called a composite type) is a data type that is built by combining one or more basic (atomic) data types into a single unit. Structured types allow related pieces of data to be grouped together and treated as a single entity, which makes programs easier to design and understand. Arrays, structures (records), and unions are common examples of structured types found in programming languages.

Difference between structured types and data structures:

A structured type is a language-level construct (like a struct or array declared in a program), whereas a data structure is a broader concept describing how data is organized logically and physically to support certain operations — a data structure may be built using one or more structured types along with algorithms that operate on them.

- Structured types are the building blocks (arrays, records/structs, unions).
- Data structures combine structured types with rules and algorithms (e.g., a linked list uses a structure/record containing data and a pointer).
- Data structures can be linear (array, stack, queue, linked list) or non-linear (tree, graph).

1.4 Atomic Type

An atomic type (also called a primitive or elementary type) is a data type that cannot be broken down further into smaller, meaningful components using the built-in features of the language. It represents a single, indivisible value. Atomic types form the foundation upon which all structured and composite data types are built.

Common atomic types:

- Integer — represents whole numbers (e.g., int in C/C++/Java).
- Float / Double — represents real numbers with a fractional part.
- Character — represents a single symbol, letter, or digit.
- Boolean — represents a logical value: true or false.

Because atomic types are indivisible, operations on them (such as addition on integers or comparison on booleans) are treated as single, basic operations by the language and are typically supported directly by the hardware or the language runtime.

1.5 Difference between Abstract Data Types, Data Types and Data Structures

Although the terms Abstract Data Type, Data Type, and Data Structure are closely related and often used together, they refer to distinct concepts at different levels of abstraction. Understanding the difference between them is fundamental to studying data structures.

Basis	Data Type	Abstract Data Type (ADT)	Data Structure
Definition	A classification specifying the kind of value a variable can hold.	A logical/mathematical model defined by its behaviour (operations), not implementation.	An actual physical/logical implementation used to organize and store data.
Focus	What kind of data is stored.	What operations can be performed.	How data is stored and operations are implemented.
Implementation detail	Provided by the language (built-in).	Hidden from the user.	Fully specified and visible to the programmer.
Examples	int, float, char, boolean	Stack, Queue, List, Map (as concepts)	Array-based stack, Linked list, Binary tree
Level	Lowest level (basic building block).	Conceptual/logical level.	Concrete/physical implementation level.

1.6 Data Types

A data type is a classification that specifies which kind of value a variable can hold and what operations can be performed on that value. Data types tell the compiler or interpreter how much memory to allocate and how to interpret the bit pattern stored in that memory. Data types are broadly classified into primitive (built-in) and non-primitive (derived/user-defined) types, and can also be classified as linear or non-linear based on how the elements are organized.

Broad classification of data types:

- Primitive Data Types — basic, built-in types such as int, float, char, and boolean.
- Non-Primitive Data Types — derived from primitive types, such as arrays, structures, unions, linked lists, stacks, queues, trees, and graphs.

1.7 Linear Data Type

A linear data structure is one in which data elements are arranged in a sequential order, and each element (except the first and the last) is connected to exactly one previous element and one next element. Because of this sequential arrangement, elements can be traversed in a single run — either from the beginning to the end or vice versa. Linear data structures are relatively simple to implement and are widely used for storing ordered data.

Examples of linear data structures:

- Array — a collection of elements of the same type stored in contiguous memory locations.
- Stack — a linear structure that follows the LIFO (Last In, First Out) principle.

- Queue — a linear structure that follows the FIFO (First In, First Out) principle.
- Linked List — a sequence of nodes where each node contains data and a reference (pointer) to the next node.

Key characteristics:

- Elements are stored and accessed in a sequential/linear manner.
- Memory utilization can be either static (array) or dynamic (linked list).
- Relatively easier to implement and requires single-level traversal.

1.8 Non-Linear Data Type

A non-linear data structure is one in which data elements are not arranged sequentially; instead, each element may be connected to multiple other elements, forming a hierarchical or networked relationship. Unlike linear structures, non-linear structures cannot be traversed in a single run and generally require more complex traversal algorithms.

Examples of non-linear data structures:

- Tree — a hierarchical structure consisting of nodes connected by edges, with one root node and multiple child nodes (e.g., binary tree, binary search tree).
- Graph — a collection of nodes (vertices) connected by edges, used to represent networks such as social networks, maps, and communication systems.

Key characteristics:

- Elements may have multiple predecessors and/or successors.
- Better suited for representing hierarchical or interconnected data.
- Traversal is more complex (e.g., BFS, DFS for graphs; in-order, pre-order, post-order for trees).

1.9 Primitive Data Type

Primitive data types are the most basic, built-in data types provided directly by a programming language. They are not derived from any other data type and serve as the fundamental building blocks for constructing more complex, non-primitive data types. Primitive types are directly supported by the underlying hardware, which makes operations on them fast and efficient.

Common primitive data types:

- Integer (int) — stores whole numbers, positive or negative.
- Floating point (float, double) — stores real numbers with decimal points.
- Character (char) — stores a single character.
- Boolean (bool) — stores a logical value, true or false.

1.10 Non-Primitive Data Type

Non-primitive data types are derived from primitive data types and are used to store a collection of values, either of the same type or of different types. Unlike primitive types, non-primitive types are not built directly into the hardware; instead, they are constructed using programming language constructs and can be manipulated through defined operations.

Common non-primitive data types:

- Array — a fixed-size collection of elements of the same data type stored contiguously.
- Structure/Record — a collection of variables of different data types grouped under a single name.
- Union — similar to a structure, but all members share the same memory location.
- Linked List, Stack, Queue, Tree, Graph — dynamic data structures built using pointers/references.

1.11 Refinement Stages

Refinement stages refer to the systematic, step-by-step process of transforming an abstract problem or an Abstract Data Type (ADT) into a fully working, concrete implementation using a programming language. This process is often called stepwise refinement or top-down design, and it ensures that complex problems are broken down into smaller, manageable, and clearly defined sub-problems before actual coding begins.

The idea behind refinement is that a problem is first described at a high level of abstraction (what needs to be done), and then gradually refined through successive stages into a low level of detail (how it will be done), until it can be directly translated into program code.

Typical stages of refinement:

1. Problem Definition — clearly stating the problem and the requirements to be solved.
2. Abstract Data Type (ADT) Specification — defining the data and operations at a logical level, without worrying about implementation.
3. Data Structure Selection — choosing an appropriate data structure (array, linked list, tree, etc.) to implement the ADT efficiently.
4. Algorithm Design — designing the step-by-step procedure (algorithm) for each operation defined in the ADT.
5. Implementation — translating the algorithm and data structure into actual program code using a chosen programming language.
6. Testing and Verification — checking the implementation for correctness, efficiency, and edge cases.

This refinement process allows programmers to manage complexity, ensures a clear separation between design and implementation, and makes it easier to modify or replace the underlying data structure later without changing the higher-level logic of the program.

Multiple Choice Questions (MCQs)

1. Which of the following is NOT a linear data structure?

A) Array
B) Stack
C) Queue
D) Tree

2. An Abstract Data Type (ADT) is defined by:

A) Its memory address
B) Its values and the operations performed on it
C) The programming language used
D) The hardware architecture

3. Which data type cannot be broken down further into smaller components?

A) Structure
B) Array
C) Atomic type
D) Union

4. Stack follows which principle?

A) FIFO
B) LIFO
C) Random access
D) Priority based

5. Queue follows which principle?

A) LIFO
B) FIFO
C) LILO
D) Random access

6. Which of the following is a non-linear data structure?

A) Linked List
B) Array
C) Graph
D) Stack

7. Which of the following is an example of a primitive data type?

A) Array
B) Structure
C) Integer
D) Linked list

8. A structure in C is an example of:

A) Primitive data type
B) Non-primitive data type
C) Atomic type
D) Abstract data type only

9. The main advantage of an ADT is:

A) Faster hardware execution
B) Hiding implementation details from the user
C) Reducing memory size
D) Increasing code length

10. Which is the correct order in stepwise refinement?

A) Implementation, ADT specification, Problem definition
B) Problem definition, ADT specification, Data structure selection, Algorithm design, Implementation
C) Algorithm design, Problem definition, Implementation
D) Testing, Implementation, Problem definition

11. A Stack ADT can be implemented using:

A) Only an array
B) Only a linked list
C) Either an array or a linked list
D) Only a tree

12. Which of the following best distinguishes a data structure from a data type?

A) Data structure is built-in; data type is user-defined
B) Data structure organizes data with defined operations; data type only classifies values
C) They are exactly the same concept
D) Data type is always non-linear

13. Trees and Graphs are classified as:

A) Linear data structures
B) Non-linear data structures
C) Primitive data types
D) Atomic types

14. Which of the following is a non-primitive, dynamic data structure?

A) Integer
B) Character
C) Linked List
D) Boolean

15. Data representation ultimately stores all data in a computer as:

A) Decimal digits
B) ASCII text only
C) Binary form (0s and 1s)
D) Hexadecimal code only

Answer Key

1. D 2. B 3. C 4. B 5. B 6. C 7. C 8. B 9. B 10. B 11. C 12. B 13. B 14. C 15. C

Broad / Long Answer Questions

7. Explain the concept of data representation in computer memory with suitable examples.
8. What is an Abstract Data Type (ADT)? Explain its characteristics with the help of a Stack ADT example.
9. Differentiate between Data Type, Abstract Data Type, and Data Structure with a comparative table.
10. What are structured types? How do they differ from a data structure? Explain with examples.
11. Define atomic type. List and explain the common atomic data types used in programming.
12. Distinguish between linear and non-linear data structures with suitable examples of each.
13. What are primitive and non-primitive data types? Explain the differences with examples.
14. Explain the stepwise refinement process used to convert an abstract problem into a working program.
15. Discuss why the choice of an appropriate data structure is important for the efficiency of a program.
16. Explain, with examples, how a single ADT (such as a Queue) can be implemented using more than one data structure.