# DATA STRUCTURE

Subject Teacher: Navjot Kaur

Class: Msc IT 2nd & LE

## Introduction

- Data structures are fundamental concepts in computer science that enable efficient storage, organization, and manipulation of data in a computer's memory or storage.
- They provide a way to manage and represent different types of data, whether simple or complex, in a way that facilitates various operations and optimizations in programming.
- There are two main categories of data structures: primitive data structures and non-primitive data structures.
  - **Primitive Data Structures**
  - **Non-Primitive Data Structures**

## Primitive Data Structures

- Primitive data structures are used to create more complex data structures. These are the basic building blocks of data manipulation. They include:

**Integer**: Stores whole numbers.

**Floating-point**: Stores numbers with decimal points.

**Character**: Stores individual characters or symbols.

**Boolean**: Stores true or false values.

## 1. Integers

**Integers** are used to represent whole numbers without any decimal points. They are commonly used for counting, indexing, and arithmetic operations. In most programming languages, integers can be classified into different sizes, such as:

- **Byte**: Typically 8 bits, representing values from -128 to 127 (signed) or 0 to 255 (unsigned).
- **Short**: Usually 16 bits, with a range of -32,768 to 32,767 (signed) or 0 to 65,535 (unsigned).
- **Int**: Often 32 bits, covering a range of approximately -2 billion to 2 billion (signed) or 0 to 4 billion (unsigned).
- **Long**: Typically 64 bits, providing a wide range for very large integers.

## 2. Floating-Point Numbers

- Floating-point numbers are used to represent real numbers, including those with decimal points or exponential notation. They are divided into two main categories:
- **Single-precision**: Also known as "float," uses 32 bits to store a floating-point number.
- **Double-precision**: Often referred to as "double," uses 64 bits and provides higher precision for decimal numbers.
- Floating-point numbers have limitations due to the way they are represented in binary, leading to issues like rounding errors and inaccuracies in some calculations.

## 3. Characters

- Characters are used to represent individual symbols, letters, or digits.
- Each character is usually associated with a numeric code using character encoding schemes like ASCII or Unicode.
- In programming, characters are typically stored using a single byte (8 bits), which allows for a wide range of symbols to be represented.

## 4. Booleans

- Booleans represent binary values, typically true or false.
- They are often used in conditional statements and logical operations.
- Booleans require only 1 bit of storage, but in most programming languages, they are stored in a full byte for memory alignment and addressing considerations.

## Non-Primitive Data Structures

- Also known as composite data structures, these are more complex and can store multiple values of different types.
- Each type of non-primitive data structure has its own characteristics, advantages, and use cases.
- The choice of data structure depends on the specific requirements of the problem you're trying to solve and the operations you need to perform on the data. They include:
  - **Arrays**: Ordered collection of elements with a fixed size.
  - **Linked Lists**: A sequence of elements where each element points to the next one.
  - **Stacks**: A linear data structure that follows the Last-In-First-Out (LIFO) principle.
  - **Queues**: A linear data structure that follows the First-In-First-Out (FIFO) principle.
  - **Trees**: Hierarchical structures with nodes connected by edges.
  - **Graphs**: Networks of interconnected nodes.
  - **Hash Tables**: Structures that use a hash function to store and retrieve data efficiently.

## 1. Arrays

- An array is a fundamental data structure that stores a collection of elements of the same data type in a contiguous memory block.
- Each element in an array is identified by its index or position, which starts from 0 for the first element, 1 for the second, and so on.
- Arrays provide a way to organize and access a group of related data items efficiently.

**Advantages of Arrays:**

1. **Random Access:** Elements in an array can be accessed directly using their indices. This allows for efficient random access and retrieval of elements.
2. **Memory Efficiency:** Arrays store elements in a contiguous memory block, which leads to efficient memory utilization and cache utilization by reducing memory fragmentation.
3. **Simplicity:** Arrays are conceptually simple to understand and implement, making them a fundamental building block for many more complex data structures.
4. **Predictable Performance:** Array access time is constant, O(1), for random access due to the direct indexing of elements.

# Contd.

**Limitations of Arrays:**

1. **Fixed Size:** Arrays have a fixed size, which is determined when the array is created. This means that you need to know the maximum number of elements in advance, and if you exceed this size, you might need to create a new, larger array and copy elements over.
2. **Wasted Memory:** If you allocate more space than you actually need for an array, you can end up wasting memory. Conversely, if you allocate too little space, you might run out of space for your data.
3. **Insertion and Deletion:** Inserting or deleting an element in the middle of an array can be inefficient, as it requires shifting all subsequent elements to make space or close gaps.
4. **Contiguous Memory Requirement:** Arrays require contiguous memory blocks, which can lead to memory fragmentation in scenarios where memory is allocated and deallocated dynamically.
5. **Lack of Flexibility:** Once an array is created, its size and type are fixed. If you need to change the size dynamically or store elements of different types, arrays might not be the best choice.

**Note:** Despite these limitations, arrays are widely used and serve as the basis for more complex data structures. In situations where the size of the collection is known and remains constant, arrays provide efficient and straightforward storage and access mechanisms

## 2. Lists

- A list is a linear data structure that holds a collection of elements in a specific order.
- Lists allow you to store and manage a sequence of items, and they provide methods to add, remove, and access elements.
- Lists are widely used in programming to represent various types of data, such as numbers, strings, or custom objects.

**Types of Lists: Arrays vs. Linked Lists**

- **1. Arrays:**
- Arrays store elements in contiguous memory locations.
- Elements can be accessed using an index.
- Constant-time access (O(1)) for random access.
- Fixed size: the size is determined at creation and cannot be changed without creating a new array.

## Contd.

- **2. Linked Lists:**
- Linked lists store elements as nodes, each containing the actual data and a reference to the next node.
- No requirement for contiguous memory allocation.
- Accessing elements requires traversing the list, resulting in linear-time access (O(n)) for random access.
- Dynamic size: nodes can be added or removed dynamically, allowing for flexible resizing.

## Contd.

**Advantages of Lists:**

1. **Dynamic Sizing:** Linked lists can dynamically grow or shrink as elements are added or removed, making them suitable for scenarios with varying data sizes.
2. **Efficient Insertions and Deletions:** Linked lists excel at inserting and deleting elements within the list, as they only require adjusting the references to neighboring nodes.
3. **Memory Efficiency:** Linked lists allocate memory as needed, reducing memory waste compared to arrays, which might reserve more space than necessary.

- **Example Scenarios:**

1. **Arrays:**
   1. Storing grades in a classroom where the number of students is fixed.
   2. Representing a deck of playing cards.
   3. Implementing a simple stack data structure.
2. **Linked Lists:**
   1. Implementing a to-do list that allows easy addition and removal of tasks.
   2. Representing a sequence of actions in a video game.
   3. Building a symbol table in a compiler.

## Stacks

- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. Stacks can be visualized as a stack of plates, where you can only add or remove plates from the top.
- In a stack, two primary operations are performed:

1. **Push:** Adding an element to the top of the stack.
2. **Pop:** Removing and returning the top element from the stack.

- Additionally, there's a third operation: 3. **Peek (or Top):** Viewing the top element without removing it.

## Contd.

- **Advantages of Stacks:**

1. **Simple and Efficient:** Stacks are simple to implement and understand, making them a convenient choice for various applications.
2. **Function Calls:** Stacks are used to manage function calls and their local variables. When a function is called, its local variables are pushed onto the stack, and when the function returns, they are popped.
3. **Undo Operations:** Stacks can be used to implement undo functionality in applications, such as text editors or graphic design software.
4. **Expression Evaluation:** Stacks are essential for evaluating arithmetic expressions, converting between infix, postfix, and prefix notations.
5. **Backtracking Algorithms:** Stacks are useful in backtracking

## Contd.

- **Disadvantages of Stacks:**

1. **Limited Access:** Accessing elements in the middle of the stack requires popping elements off until the desired element is reached. This can be inefficient for large stacks.
2. **Fixed Size:** Some implementations of stacks have a fixed size, which can lead to stack overflow errors if the size is exceeded.
3. **No Random Access:** Unlike arrays, stacks do not support direct access to elements based on an index.
4. **Dynamic Memory Allocation:** If implemented using dynamic memory allocation (linked list), the overhead of managing memory can be higher compared to arrays.
5. **Not Suitable for All Problems:** While stacks are versatile, they are not the best choice for all scenarios. Some problems might be better solved with other data structures.

## Contd.

- **Example Scenarios:**

1. **Function Call Management:** Tracking function calls and local variables in programming languages.
2. **Undo/Redo Functionality:** Implementing undo and redo actions in applications.
3. **Expression Parsing and Evaluation:** Converting and evaluating mathematical expressions.
4. **Backtracking Algorithms:** Solving mazes, puzzles, and finding paths in graphs.

## Queues

- A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. This means that the first element added to the queue is the first one to be removed.
- Queues can be visualized as a line of people waiting in line, where the person at the front is the next to be served.
- In a queue, two primary operations are performed:

1. **Enqueue:** Adding an element to the back (rear) of the queue.
2. **Dequeue:** Removing and returning the front element from the queue.

Additionally, there's a third operation:

3. **Front (or Peek):** Viewing the front element without removing it.

## Contd.

- **Advantages of Queues:**

1. **Order Preservation:** Queues maintain the order in which elements are added, making them useful for scenarios where order matters.
2. **Task Scheduling:** Queues are used in task scheduling algorithms to manage tasks in the order they were received.
3. **Breadth-First Search:** Queues are essential for implementing breadth-first search algorithms in graph traversal.
4. **Print Queue:** Managing print jobs in a printer queue, ensuring they are processed in the order they were submitted.
5. **Buffering:** Queues are used for buffering data between different parts of a system, ensuring smooth data flow.

## Contd.

- **Disadvantages of Queues:**

1. **Limited Access:** Accessing elements in the middle of the queue requires dequeuing elements until the desired element is reached. This can be inefficient for large queues.
2. **Fixed Size:** Some implementations of queues have a fixed size, which can lead to queue overflow errors if the size is exceeded.
3. **No Random Access:** Like stacks, queues do not support direct access to elements based on an index.
4. **Dynamic Memory Allocation:** If implemented using dynamic memory allocation (linked list), the overhead of managing memory can be higher compared to arrays.
5. **Not Suitable for All Problems:** While queues are versatile, they might not be the best choice for all scenarios. Some problems might be better solved with other data structures.

## Contd.

**Example Scenarios:**

1. **Task Scheduling:** Managing a queue of tasks to be processed by a CPU scheduler.
2. **Breadth-First Search:** Traversing graphs in breadth-first order to find the shortest path.
3. **Print Queue:** Managing a queue of print jobs in a printer system.
4. **Buffering:** Buffering data between input and output devices to ensure smooth data flow.