Faculty of Information Technology
University of Moratuwa

BSc (Hons) in Information Technology
BSc (Hons) in Artificial Intelligence

IN 1111 – Data Structures & Algorithms I

Lecture 01

Mr. Nadana Swasthi
Lecturer (On Contract)
nadanas@uom.lk

Outline of Syllabus

- Why data structures and algorithms?
- Data structures
  - Arrays, Stacks, Queues, Linked lists
- Sorting algorithms
  - Insertion sort, Selection sort, Bubble sort
- Recursion
- Applications of data structures and algorithms
- Algorithm complexity

Recommended text

1. Cormen, T., Leiserson, C., Stein, C., & Rivest, R. (2009). ***Introduction to Algorithms. 3rd Edition***. MIT Press. ISBN-13: 978-0262033848, ISBN-10: 9780262033848.

2. Gilberg, R., & Forouzan, B. (2004). Data Structures***: A Pseudocode Approach with C. 2nd Edition. Cengage Learning***. ISBN-10 : 0534390803, ISBN-13 : 978-0534390808.

Lesson Plan

Week 01	Intoduction to Data Structures & Arrays	
Week 02	Pseudocode & Linked List	
Week 03	Linked List	
Week 04	Doubly Linked List	
Week 05	Circular Linked List	
Week 06	Queue	
Week 07	Stack	
Week 08	Recursion & Sorting Algorithms Intro	Quiz 01
Week 09	Insertion Sort	
Week 10	Selection Sort	
Week 11	Bubble Sort	
Week 12	Applications of DSA	
Week 13	Algorithm Complexity	Quiz 02

80% OF ATTENDANCE IS MANDATORY!

Introduction

A ***data structure*** is a specialized format for **organizing** and **storing** data. Data structure is designed to organize data to suit a **specific purpose** so that it can be accessed and worked with in appropriate ways.

An ***algorithm*** is a **set of step-by-step instructions** to solve **a given problem** or achieve a **specific goal**.

Data Structures and Algorithms (DSA) is about finding efficient ways to store and retrieve data, to perform operations on data, and to solve specific problems.

By understanding DSA, you can:

- Decide which data structure or algorithm is best for a given situation.
- Make programs that run faster or use less memory.
- Understand how to approach complex problems and solve them in a systematic way

Data Structures

- A **data structure** is a way to organize and store data so that we can **use it efficiently**.
- The choice of data structure affects **speed, memory, and simplicity** of solving a problem.
- An **algorithm** is a **step-by-step procedure to solve a problem**. Data structures store data, and algorithms **act on it**.

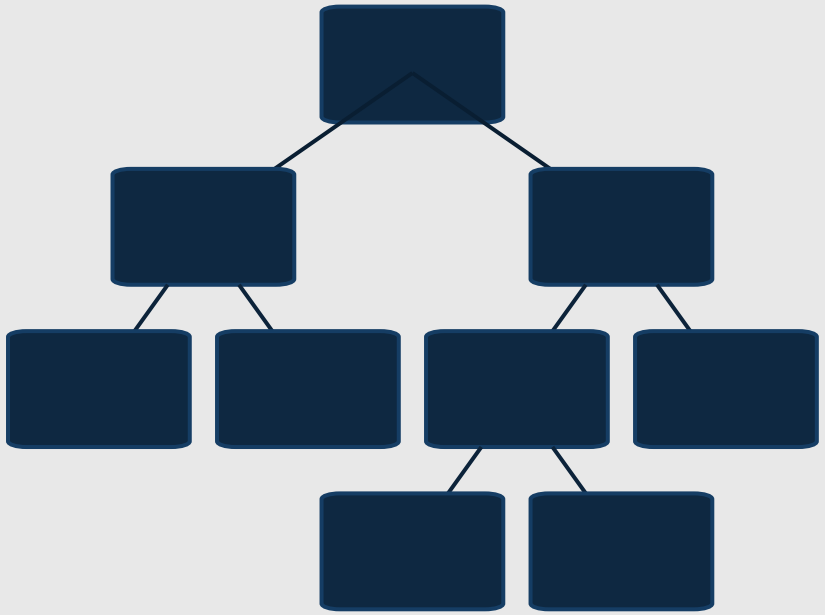
    - Student marks in a class
    - Calendar dates in a month
    - Music playlist
    - Browser
    - CPU task scheduling
    - Customer service lines

Data Structures

Linear Data Structures

Data elements are organized sequentially and therefore they are easy to implement in the computer's memory

Arrays, Linked lists, Queue, Stack



Non-linear Data Structures

A data element can be attached to several other data elements to represent specific relationships that exist among them.

One element can be connected to more than two adjacent element or arranged in a hierarchical manner.

Trees, Graphs, Heaps

Data Structures

Data structure can be classified into two types according to the fixed or modified data size

Static Data Structures

- Consists of a number of determined elements (not increased nor decreased).
- You must provide the required and sequential number of memory locations.
- The number of locations is fixed not changed.
- There is no a good usage of memory.
- The processing is slow spatially with adding and deleting, because it needs shifting operation.

Dynamic Data Structures

- The data size is changed.
- Use the pointers.
- The number of locations is changed not fixed; it depends only on the exist memory size.
- The usage of memory is better, because the locations must not be sequential.
- The speed of processing is high

Arrays Data Structure

An **array** is a **collection of fixed number of elements of the same type** stored in **contiguous memory locations**. Each element can be accessed using an **index.**

Following are the important terms to understand the concept of Array.

- **Element:** Each item stored in an array is called an element.
- **Index:** Each location of an element in an array has a numerical index, which is used to identify the element.

Declaring an array means specifying the following:

1. **Data type:** The kind of values it can store, for example, int, char, float, double.
2. **Name:** Reference to Identify the array.
3. **Size**: The maximum number of values that the array can hold.

```
int  num [10];
```

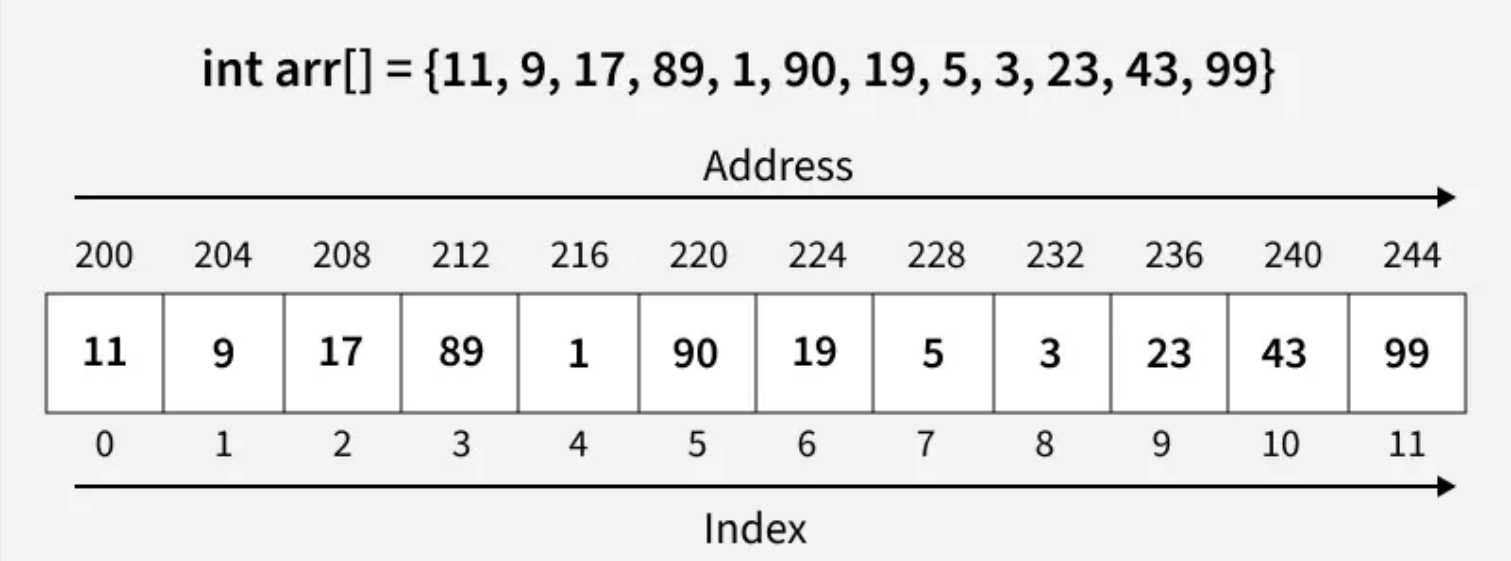
Memory Allocation

Contiguous Memory

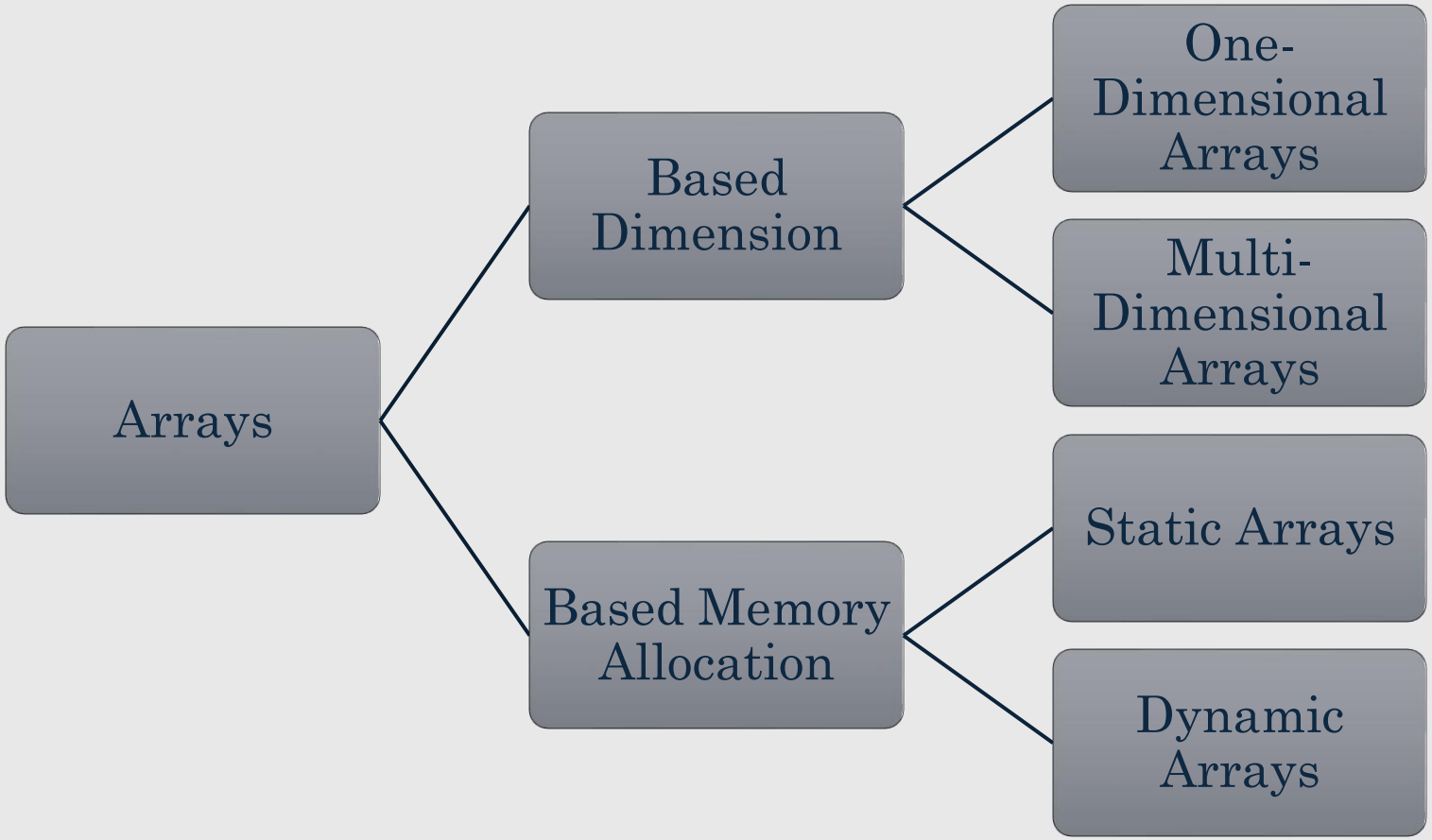
- All array elements are stored **one after the other** in memory and this allows **direct access** to any element using its **index**.

Address calculation

Address of arr[i] = Base_address + (i * size_of_element)



Types of Arrays



Basic Operations of Arrays

Traverse

Print/visit all the array elements one by one.

Insertion

Add an element at the given index/position .

Deletion

Delete an element at the given index/position.

Search

Search an element using the given index or by the value.

Update

Update an element at the given index.

Traverse

```
PROCEDURE TRAVERSE(Array : ARRAY OF INTEGERS, Size : INTEGER)
BEGIN
    DECLARE Index : INTEGER
    FOR Index ← 0 TO Size - 1 DO
        PRINT Array[Index]
    END FOR
END
```

What happens

- Start from index **0**
- Move one element at a time
- Access and print each element
- Stop at index **n–1**

Search

```
PROCEDURE SEARCH_VALUE(Array : ARRAY OF INTEGERS, Size : INTEGER, Key : INTEGER)
BEGIN
    DECLARE Index : INTEGER
    DECLARE Found : BOOLEAN
    Found ← FALSE

    FOR Index ← 0 TO Size - 1 DO
        IF Array[Index] = Key THEN
            PRINT "Found at index " + Index
            Found ← TRUE
            BREAK              // Stop after first occurrence
        END IF
    END FOR

    IF Found = FALSE THEN
        PRINT "Element not found"
    END IF
END
```

Update

1. Update by Index

- You know the **position** of the element in the array.
- Simply replace the old value with the new value at that index.

```
PROCEDURE UPDATE(Array : ARRAY OF INTEGERS, Index : INTEGER, Value : INTEGER)
BEGIN
    Array[Index] ← Value
END
```

2. Update by Element Value

- You know the **value** you want to change but not its index.
- Traverse the array to find the element.
- Replace the first occurrence with the new value (or all occurrences if required).
- If the element is not found, display a message.

Update

Exercise

You are given an array of integers. Write pseudocode to **update a given element in the array** with a new value.

Input:

- An array of integers Array of size n
- The value to be updated OldValue
- The new value NewValue

Output:

- The updated array print ‘Array is updated’
- If the OldValue is not found, print "Element not found"

Update

```
PROCEDURE UPDATE_BY_VALUE(Array : ARRAY OF INTEGERS, Size : INTEGER, OldValue : INTEGER, NewValue : INTEGER)
BEGIN
    DECLARE Index : INTEGER
    DECLARE Found : BOOLEAN
    Found ← FALSE

    FOR Index ← 0 TO Size - 1 DO
        IF Array[Index] = OldValue THEN
            Array[Index] ← NewValue
            Found ← TRUE
            PRINT "Array updated successfully"
            BREAK           // Stop after first match
        END IF
    END FOR

    IF Found = FALSE THEN
        PRINT "Element not found"
    END IF
END
```

Insert

Insertion is the process of adding a new element to an array at a specific position.

There are three main scenarios:

1. **Insert at the front:** Add the element at the beginning of the array.
2. **Insert at the back:** Add the element at the end of the array.
3. **Insert at any position:** Add the element at a specific index in the array.

Important:

- When inserting at front or in between, elements may need to be **shifted** to make space.
- Inserting at the back does not require shifting.

Insert at the front

```
PROCEDURE INSERT_AT_FRONT(Array : ARRAY OF INTEGERS, Size : INTEGER, Value : INTEGER)
BEGIN
    DECLARE Index : INTEGER

    // Shift elements one position to the right to make space at index 0
    FOR Index ← Size - 1 TO 0 STEP -1
        Array[Index + 1] ← Array[Index]
    END FOR

    // Insert new value at the front
    Array[0] ← Value

    // Update logical size of the array
    Size ← Size + 1

    PRINT "Element inserted at front successfully"
END
```

Insert at the back

```
PROCEDURE INSERT_AT_BACK(Array : ARRAY OF INTEGERS, Size : INTEGER, Value : INTEGER)
BEGIN
    // Insert new value at the end
    Array[Size] ← Value

    // Update logical size of the array
    Size ← Size + 1

    PRINT "Element inserted at back successfully"
END
```

Insert at any middle position:

```
PROCEDURE INSERT_AT_POSITION(Array : ARRAY OF INTEGERS, Size : INTEGER, Position : INTEGER, Value : INTEGER)
BEGIN
    DECLARE Index : INTEGER

    // Shift elements to the right starting from the end up to Position
    FOR Index ← Size - 1 TO Position STEP -1
        Array[Index + 1] ← Array[Index]
    END FOR

    // Insert new value at the specified position
    Array[Position] ← Value

    // Update logical size of the array
    Size ← Size + 1

    PRINT "Element inserted at position " + Position + " successfully"
END
```

- Can this pseudocode correctly insert an element at the back of the array?
- If not, suggest an improved version that handles insertion at the back, front, and middle correctly.

Insert at any position

```
PROCEDURE INSERT_AT(Array : ARRAY OF INTEGERS, Size : INTEGER, Position : INTEGER, Value : INTEGER)
BEGIN
    DECLARE Index : INTEGER

    IF Position = Size THEN
        // Insert at back
        Array[Size] ← Value
    ELSE
        // Shift elements to the right
        FOR Index ← Size - 1 TO Position STEP -1
            Array[Index + 1] ← Array[Index]
        END FOR

        // Insert at the specified position
        Array[Position] ← Value
    END IF

    // Update logical size
    Size ← Size + 1

    PRINT "Element inserted at position " + Position + " successfully"
END
```

Homework

What happens when an array becomes almost full or completely full?
Explain the limitations and what options a programmer has at that point.

Thank you