Introduction to Computer Programming

- Week 5&6

- *Eng. Sylvain Manirakiza* -

Sun - October 12, 2025

Chap 3. Linear Structures

- Week 5

- *Eng. Sylvain Manirakiza -*

Sun - Oct 12, 2025

Linear Structures

linear structures are essential for organizing and manipulating data efficiently.

Today's Objectives

Objective 1	• Understand what arrays are and their types.
Objective 2	• Learn how to perform key operations on arrays.
Objective 3	• Explore different sorting algorithms and their efficiencies.
Objective 4	• Implement sorting techniques in real-world examples.

Introduction to Linear Structures

What are Data Structures?

Data structure is:

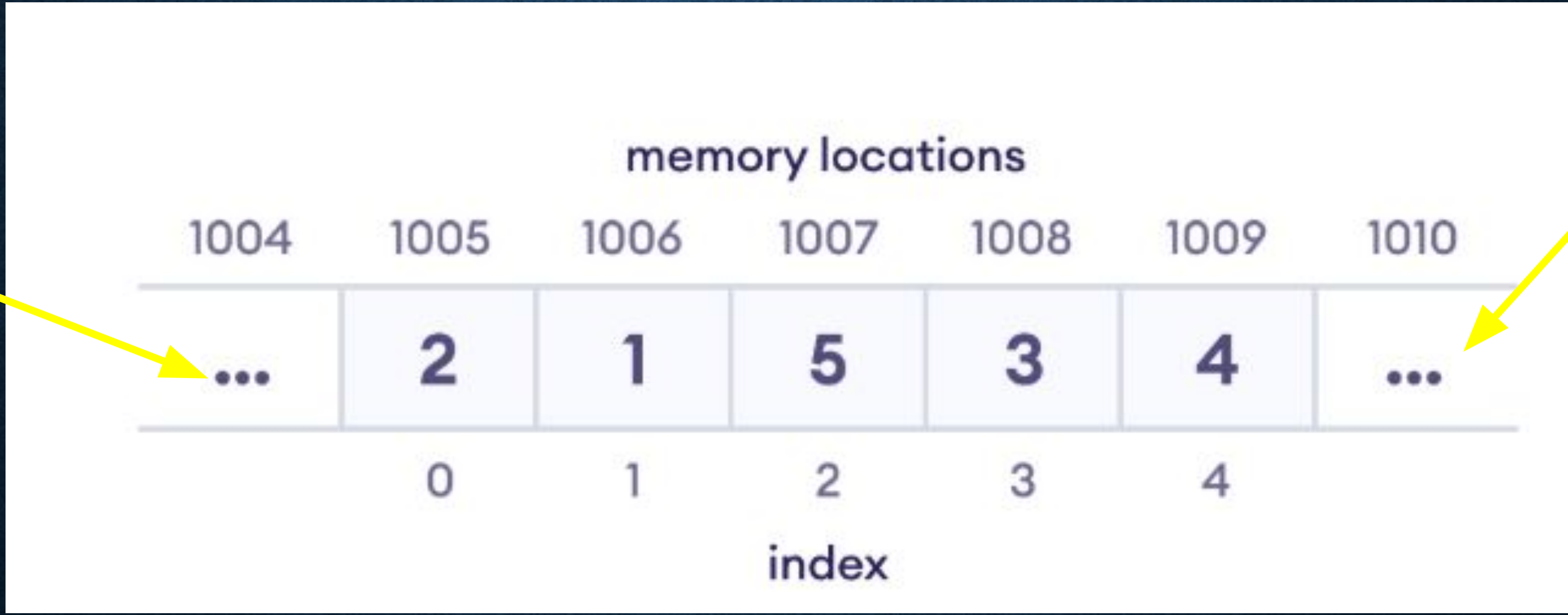
- It is a way of arranging data on a computer so that it can be accessed and Processed/updated efficiently.

What is Linear Structures?

A linear structure is a data structure where elements are arranged sequentially, and each element has a unique predecessor and successor (except the first and last).

Introduction to Linear Structures - Arrays

First



Second

Array data Structure Representation

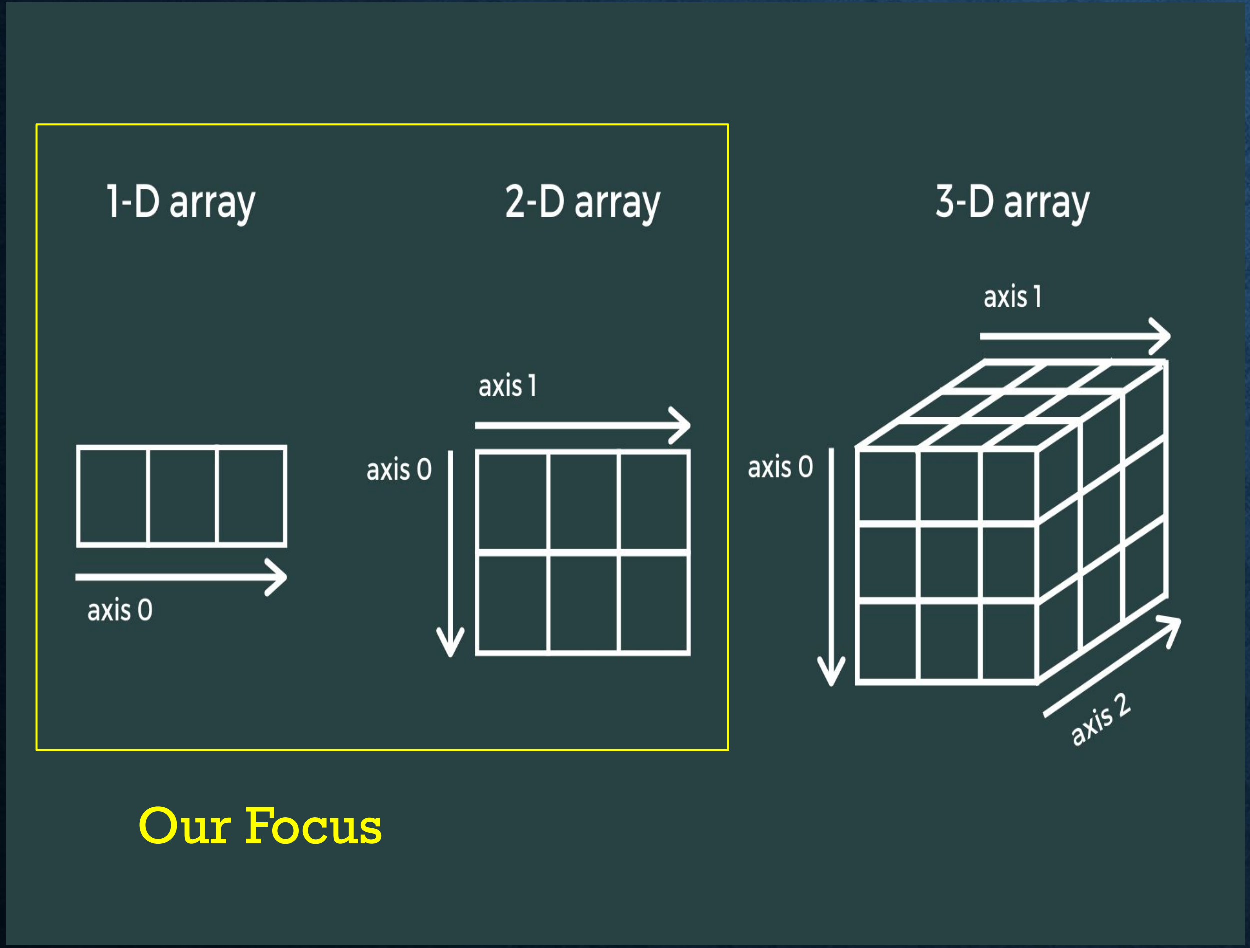
Importance of Linear Structures:

1. **Efficient storage and retrieval**
2. **Basis for advanced data structures (trees, graphs)**
3. **Used in memory allocation, scheduling, and data processing**

Types of Linear Structures

1. **Arrays** – Fixed-size sequential storage.
2. **Linked Lists** – Dynamic memory allocation.
3. **Stacks** – Last In, First Out (LIFO) structure.
4. **Queues** – First In, First Out (FIFO) structure.

Arrays



An array is a collection of elements stored in contiguous memory locations.

Arrays

Why Use Array?

✔Efficient data storage

✔ Quick access using an index

✔ Useful for handling multiple values of the same type

Example : Array Declaration

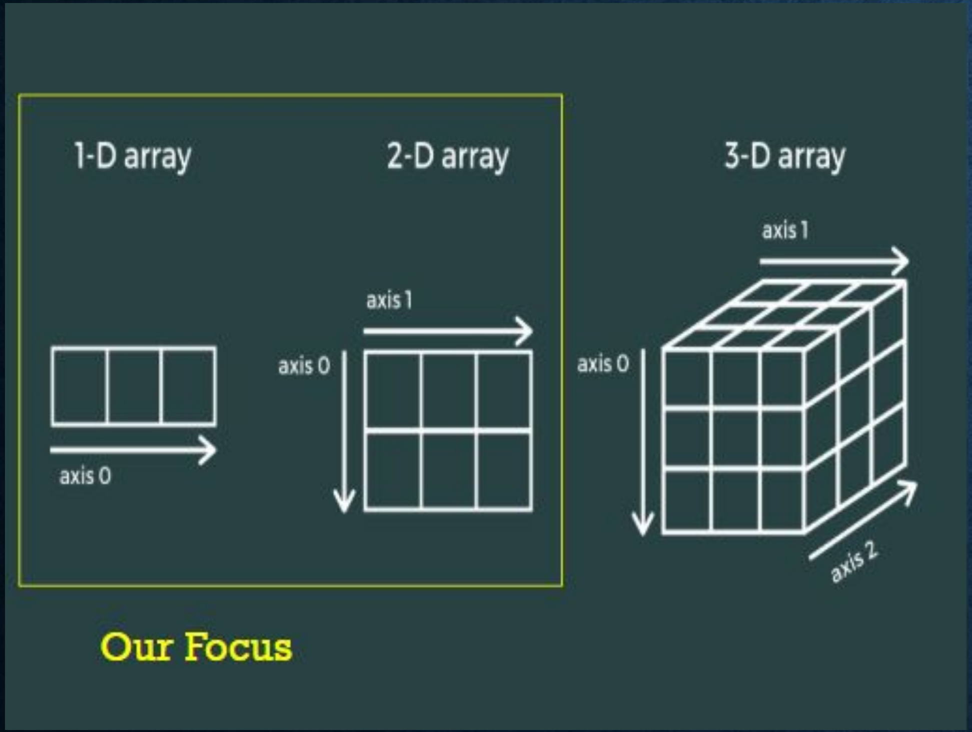
```
Variable marks as Integer[5]
marks ← {10, 20, 30, 40, 50}
Write marks[2]

// Output: 30
```

Index	Value
marks[0]	10
marks[1]	20
marks[2]	30
marks[3]	40
marks[4]	50

Write marks[2]: Retrieves the value at index 2, which is 30.

1. Types of Arrays



Type	Description	Example
1D Array (One-Dimensional)	A list of elements stored in a single row	`numbers = [10, 20, 30, 40]`
2D Array (Two-Dimensional)	A table-like structure with rows and columns	`matrix = [[1, 2], [3, 4]]`
Multi-Dimensional Array	Arrays with more than two dimensions (3D, 4D, etc.)	`cube[3][3][3]`

2. Operations on Arrays:

Operations	**Description**
Insertion	Add an element at a specific index
Deletion	Remove an element from an index
Searching	Find an element (Linear Search, Binary Search)
Traversal	Access each element in sequence(One by one)
Update	Adjust or assign a new value of a given index

2. 1. Insert

Insert 25 at index 2

- Numbers = **[10, 20, 30, 40, 50]**
- **Steps:**
  - Shift elements to the right from the insertion point.
  - Insert the new value at the specified index.
- Results → [10, 20, **25**, 30, 40, 50]

2. 1. Insert - Pseudocode

```
Start
Declare Array[10] as Integer
Input n            // number of current elements
Input position, value

For i ← n DownTo position
    Array[i + 1] ← Array[i]     // Shift elements right
End For

Array[position] ← value
n ← n + 1

Write "Element inserted successfully"
End
```

2. 2. Delete

Delete element at index 2

- Array = **[10, 20, 30, 40, 50]**
- **Steps:**
  - Shift elements to the left from the deletion point.
  - Reduce the size of the array by 1.
- Results → [10, 20, ~~30,~~ 40, 50]
- **[10, 20, 40, 50]**

2. 2. Delete - Pseudocode

```
Start
Input position
For i ← position To n - 1
    Array[i] ← Array[i + 1]      // Shift left
End For
n ← n - 1
Write "Element deleted successfully"
End
```

2. 3. Search

The **Search Operation** finds the index of a specific value in the array. If the value is not found, it returns an error or a special value (e.g., 30).

- ➔ Array = **[10, 20, 30, 40, 50]**
- ➔ **Search for the Value 30**
- ➔ Results → 2 (index of 30 in the array)

There are two types of Searching:

1. Linear Search (Check if a value exists in the array and it checks each element one by one and stops when found).
2. Binary Search (Efficient for sorted arrays - The array must be sorted (e.g. [2, 4, 6, 8, 10]).).

2. 3. 1 Linear Search -Pseudocode

```
Start
Input value
found ← False

For i ← 0 To n - 1
    If Array[i] = value Then
        Write "Element found at position ", i
        found ← True
        Exit For
    End If
End For

If found = False Then
    Write "Element not found"
End If
End
```

2. 3. 2 Binary Search -Pseudocode

Smart searching that skips many elements — but only works when the array is sorted.

```
Start
low ← 0
high ← n - 1
found ← False

While low ≤ high AND found = False
    mid ← (low + high) / 2
    If Array[mid] = target Then
        found ← True
    ElseIf Array[mid] < target Then
        low ← mid + 1
    Else
        high ← mid - 1
    End If
End While
End
```

2. 4. Traversal

- Traversal means visiting each element in an array(like “reading” the array), one by one, usually from the first index to the last.
- Think of it like walking through every seat in a row to see what’s there.

```
For i ← 0 to n - 1
    Write Array[i]
End For
```

You visit all elements, regardless of what they are.

2. 5. Updating

Updating means changing the value of an element at a given index/ Modify a value at a given index.

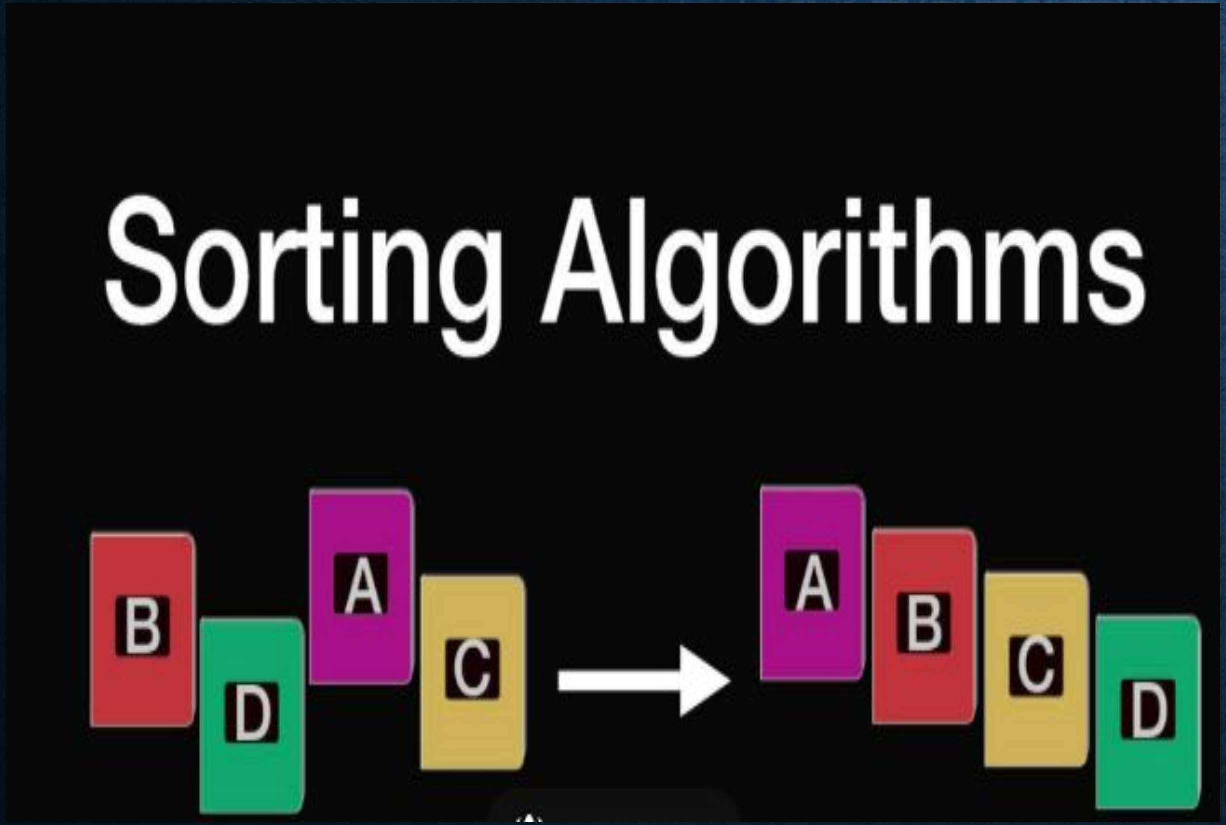
- ➔ Array = **[10, 20, 30, 40, 50]**
  - ◆ Update the value of index 2 to 99

Results → [10, 20, **99**, 40, 50]

2. 5. Updating - Pseudocode

```
Start
Input position, new_value
If position >= 0 AND position < n Then
    Array[position] ← new_value
    Write "Value updated successfully"
Else
    Write "Invalid position"
End If
End
```

- Sorting-



Sorting is the process of arranging elements in a specific order (ascending or descending).

Sorting

- ✓ Sorting is the "process through which data are arranged according to their values."

- ✓ Sorted lists allow us to search for data more efficiently.

- ✓ Sorting algorithms depend heavily on swapping elements – **remember, to swap elements, we need to use a temporary variable!**

- ✓ We'll examine three sorting algorithms – the Selection Sort, the Bubble Sort and the Insertion Sort.

Types of Sorting algorithms

There are many, many different types of sorting algorithms, but the primary ones are:

- ✓ Bubble Sort
- ✓ Selection Sort
- ✓ Insertion Sort

1. Selection Sort

- ✓ Divide the list into two sublists: sorted and unsorted, with the sorted sublist preceding the unsorted sublist.
- ✓ In each pass,
- ✓ Find the smallest item in the unsorted sublist
- ✓ Exchange the selected item with the first item in the sorted sublist.
- ✓ Thus selection sort is known as exchange selection sort that requires single array to work with.

Selection sort example

Array **numlist** contains

15 6 13 22 3 52 2

Smallest element is 2. Exchange 2 with element in 1st array position (*i.e.* element 0) which is 15

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
2	2	2	2	2	2	2
6	3	3	3	3	3	3
13	13	6	6	6	6	6
22	22	22	13	13	13	13
3	6	13	22	15	15	15
52	52	52	52	52	22	22
15	15	15	15	22	52	52

1. Selection Sort - Pseudocode

```
Function SelectionSort(arr)
  n ← Length of arr

  For i ← 0 to n-1 Do
    minIndex ← i  // Assume the first element is the smallest

    For j ← i+1 to n-1 Do
      If arr[j] < arr[minIndex] Then
        minIndex ← j  // Update minIndex if a smaller element is found
      End If
    End For

    Swap arr[i] and arr[minIndex]  // Swap the smallest element with the current
position
  End For
End Function
```

Observations about Selection sort

Advantages:

1. Easy to use.
2. The efficiency DOES NOT depend on initial arrangement of data
3. Could be a good choice over other methods when data moves are costly but comparisons are not.

Disadvantages:

1. Performs more transfers and comparison compared to bubble sort.
2. The memory requirement is more compared to insertion & bubble sort.

2. Bubble Sort

Basic Idea:-

- ✓ Divide list into sorted the unsorted sublist is "bubbled" up into the sorted sublist.
- ✓ Repeat until done:

  1. Compare adjacent pairs of records/nodes.
  2. If the pair of nodes are out of order, exchange them, and continue with the next pair of records/nodes.
  3. If the pair is in order, ignore and unsorted sublists.

- ✓ Smallest element in them and continue with the next pair of nodes.

Bubble sort example

Initial array elements

9 7 4 6 1

Array after pass-1

7 9 4 6 1

7 4 9 6 1

7 4 6 9 1

7 4 6 1 9

Array after pass-2

4 7 6 1 9

4 6 7 1 9

4 6 1 7 9

Array after pass-3

(No swap)

4 6 1 7 9

4 1 6 7 9

Array after pass-4

1 4 6 7 9

Final sorted array

1 4 6 7 9

2. Bubble Sort - Pseudocode

```
Function BubbleSort(arr)
  n ← Length of arr
  For i ← 0 to n-1 Do
    swapped ← False
    For j ← 0 to n-i-2 Do
      If arr[j] > arr[j+1] Then
        Swap arr[j] and arr[j+1]
        swapped ← True
      End If
    End For
    If swapped = False Then
      Break  // Array is already sorted
    End If
  End For
End Function
```

Observations about Bubble Sort

Advantages:

1. Easy to understand & implement.
2. Requires both comparison and swapping.
3. Lesser memory is required compared to other sorts.

Disadvantages:

1. As the list grows larger the performance of bubble sort get reduced dramatically.
2. The bubble sort has a higher probability of high movement of data.

3. Insertion

- ✓ Commonly used by card players: As each card is picked up, it is placed into the proper sequence in their hand.
- ✓ Divide the list into a sorted sublist and an unsorted sublist.
- ✓ In each pass, one or more pieces of data are removed from the unsorted sublist and inserted into their correct position in a sorted sublist.

Insertion sort program

Initially array contents are (9,4,3,2,34,1)

Pass1	9	4	3	2	34	1
Pass 2	4	9	3	2	34	1
Pass 3	3	4	9	2	34	1
Pass 4	2	3	4	9	34	1
Pass 5	2	3	4	9	34	1
Pass 6	1	2	3	4	9	34
Final Sorted array	1	2	3	4	9	34

3. Insertion - Pseudocode

```
Function InsertionSort(arr)
  n ← Length of arr
  For i ← 1 to n-1 Do
    key ← arr[i]
    j ← i - 1
    While j >= 0 AND arr[j] > key Do
      arr[j + 1] ← arr[j]  // Shift element right
      j ← j - 1
    End While
    arr[j + 1] ← key  // Insert key in correct position
  End For
End Function
```

Application

Case Study /Application

Manage marks of 5 students in a test.

- Use arrays to simulate how a school system stores, retrieves, and updates students' marks.
- The teacher wants to:
  - Store students' marks.
  - Display all marks (Traversal).
  - Find a student's mark by position (Access/Search).
  - Update a mark (Update).
  - Add a new student's mark (Insertion).
  - Remove a student's mark (Deletion).

Step 1: Declare the Array

```
 Declare Marks[10] as Integer
Input n    // number of students, e.g. 5

For i ← 0 to n - 1
   Input Marks[i]
End For
```

e.g. Marks = [78, 85, 90, 67, 88]

Step 2: Display All Marks

```
For i ← 0 to n - 1
   Write "Student", i + 1, "Mark:", Marks[i]
End For
```

- Student 1 Mark: 78
- Student 2 Mark: 85
- Student 3 Mark: 90
- Student 4 Mark: 67
- Student 5 Mark: 88

Step 3: Find a Specific Mark

```
Input target
found ← False
For i ← 0 to n - 1
    If Marks[i] = target Then
        Write "Mark found at position", i
        found ← True
        Exit For
    End If
End For
If found = False Then
    Write "Mark not found"
End If
```

Step 4:Correct a Student’s Mark

```
Input position, newMark
If position >= 0 AND position < n Then
    Marks[position] ← newMark
    Write "Mark updated successfully!"
Else
    Write "Invalid position"
End If
```

Step 5: Add a New Student’s Mark

```
Input position, newMark
For i ← n DownTo position
    Marks[i + 1] ← Marks[i]
End For
Marks[position] ← newMark
n ← n + 1
Write "New mark inserted!"
```

Step 6: Remove a Student's Mark

```
Input position
For i ← position To n - 1
    Marks[i] ← Marks[i + 1]
End For
n ← n - 1
Write "Mark deleted successfully!"
```

Homework

Write a program to Manage Marks for 5 Students in 5 Topics

End