**Module 1: Introduction & Foundations**

**Introduction**

Every computer program, no matter how simple or complex, needs to work with **data**. From a basic calculator storing two numbers, to social networks handling billions of user connections, the way we **organize and manage data** determines whether a system is **efficient or slow, reliable or fragile**.

This is where **Data Structures and Algorithms (DSA)** come in.

* A **data structure** is like a container — it defines how data is arranged in memory so it can be accessed and modified quickly.
* An **algorithm** is a recipe — it defines the steps to solve a problem using data structures.

Together, DSA is the **foundation of computer science**. Without them:

* Google would take hours to show search results.
* Facebook could not manage your friend connections.
* Banks could not handle millions of daily transactions.

In this first module, we will explore the **basic building blocks**:

* What data structures are and why they are important.
* The difference between **linear** (sequential) and **non-linear** (hierarchical) structures.
* The idea of **Abstract Data Types (ADT)**, which describe *what* a structure can do without worrying about *how*.
* The use of **Big O Notation**, which helps us measure efficiency and predict scalability.

This module is designed to give you the **foundation** you will need for the entire course. By the end, you will not only understand the concepts but also practice them with **simple C# programs** that show how performance changes depending on the data structure or algorithm used.

**Lesson Overview**

This module introduces the foundations of **Data Structures and Algorithms (DSA)**. Data structures are methods of organizing and storing data so that operations such as searching, updating, and deleting can be done efficiently. Algorithms are step-by-step instructions for solving problems.

By learning DSA, you will not only know *how to make a program work*, but also *how to make it work efficiently*.

**Learning Outcomes**

After completing this module, students should be able to:

1. Define what a **data structure** is and explain its importance in software development.
2. Differentiate between **linear** and **non-linear** data structures with real-world examples.
3. Describe the concept of **Abstract Data Types (ADT)** and identify common ADTs in programming.
4. Explain and apply **Big O Notation** to analyze the time complexity of simple algorithms.
5. Develop small C# programs to demonstrate constant-time (O(1)), linear-time (O(n)), and quadratic-time (O(n²)) operations.

**Lesson Content**

**1. Data Structures Overview & Importance**

**What is a Data Structure?**

A **data structure** is a way of arranging data in memory so that it can be used efficiently. It not only defines the format in which data is stored but also specifies the operations that can be performed.

Examples include:

* **Array** → Stores elements in a fixed-size list.
* **Linked List** → Stores elements with pointers connecting them.
* **Stack** → Last In, First Out (LIFO).
* **Queue** → First In, First Out (FIFO).
* **Tree** → Hierarchical structure (e.g., family tree).
* **Graph** → Network structure (e.g., social networks).

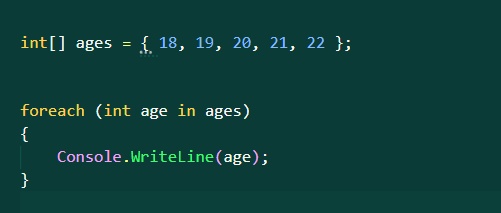
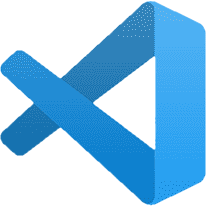
**Why are Data Structures Important?**

* **Efficiency:** Choosing the right data structure saves time and memory.
* **Scalability:** Efficient programs can handle larger inputs.
* **Maintainability:** Organized data is easier to manage and debug.

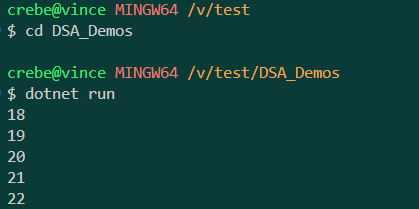
**Real-world case study:**

* Google Search uses trees and graphs to index billions of web pages.
* Banking systems use queues to process thousands of transactions per second.

*Code Walkthrough*



Output:



**What did we make:**

1. **We have a list of student ages.  
   Imagine a classroom where students are aged:**
2. **18, 19, 20, 21, 22**

**This is stored in an array. Each age has a position (index):**

* + **Position 0 → 18**
  + **Position 1 → 19**
  + **Position 2 → 20**
  + **Position 3 → 21**
  + **Position 4 → 22**

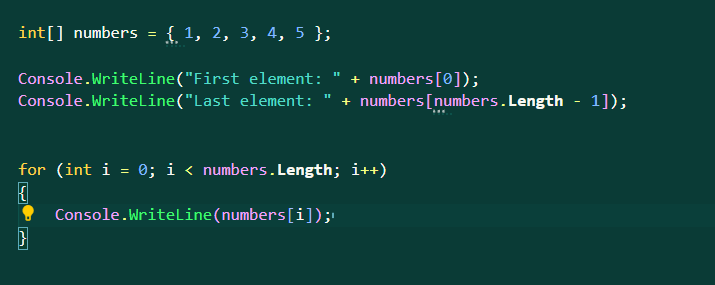
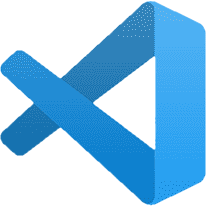
1. **We want to look at every student’s age.**
   * **Instead of asking for just the first or last, this time we want all of them.**
   * **We go through the list one by one:**
     + **First age: 18**
     + **Next age: 19**
     + **Then 20, 21, and finally 22**
2. **The process.**
   * **We don’t skip or jump.**
   * **We read every item in order.**
   * **This guarantees we see the entire group.**
3. **Efficiency (Big O).**
   * **If the class has 5 students, we make 5 checks.**
   * **If it had 100 students, we’d make 100 checks.**
   * **If it had 1,000 students, 1,000 checks.**
   * **The time grows directly with the number of items.**
   * **This is O(n) linear time.**

**2. Linear vs Non-linear Structures**

**Linear Data Structures**

* Stored in sequence, one after another.
* Examples: Arrays, Linked Lists, Stacks, Queues.
* Analogy: A line of people waiting at the grocery store.

*Code Walkthrough*



Output:



**What process happened:**

**1. int[] numbers = { 1, 2, 3, 4, 5 };**

* We create an **array** named numbers.
* Arrays in C#:
  + Have a **fixed size** (this one has 5 elements).
  + Store elements **in order**, each one assigned an **index** starting from 0.

So this array looks like:

| **Index** | **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- | --- |
| Value | 1 | 2 | 3 | 4 | 5 |

**2. Console.WriteLine("First element: " + numbers[0]);**

* numbers[0] → directly fetches the value at index 0.
* That value is 1.
* Output:
* First element: 1

⏱ **Efficiency:**

* Accessing any index in an array is **O(1) constant time**.
* Why? Because arrays are stored in continuous memory, so the program can instantly “jump” to index 0 (or any index).

**3. Console.WriteLine("Last element: " + numbers[numbers.Length - 1]);**

* numbers.Length gives the size of the array (5).
* numbers.Length - 1 = 4.
* So this fetches numbers[4], which is the last element = 5.
* Output:
* Last element: 5

**Efficiency:**

* Also **O(1) constant time** because array indexing doesn’t depend on size.

**4. The for loop**

for (int i = 0; i < numbers.Length; i++)

{

Console.WriteLine(numbers[i]);

}

* Starts at i = 0 and goes until i < numbers.Length (which is 5).
* So it runs when i = 0, 1, 2, 3, 4.
* On each run:
  + numbers[i] fetches the element at index i.
  + Console.WriteLine(numbers[i]); prints it.

**Outputs:**

1

2

3

4

5

**Efficiency:**

* The loop runs once **for every element**.
* So if the array had n elements, it would run n times.
* That makes it **O(n) linear time**.

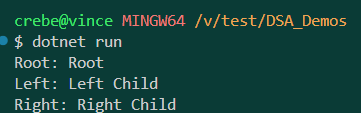
**Non-linear Data Structures**

* Data arranged hierarchically or in networks.
* Examples: Trees, Graphs.
* Analogy: A family tree

*Code Walkthrough*



Output:



Anu nga ba yung ginawa natin?

**1. Defining the Node**

class Node

{

public string data;

public Node? left, right;

public Node(string value)

{

data = value;

left = right = null;

}

}

* This is a **blueprint for a tree node**.
* Each node stores:
  + data → the value inside the node (here it’s text, like "Root").
  + left → a link to the left child.
  + right → a link to the right child.
* At first, both left and right are empty (null).

Think of it like a box that can hold a value and two arrows: one pointing left, one pointing right.

**2. Creating the Tree**

Node root = new Node("Root");

root.left = new Node("Left Child");

root.right = new Node("Right Child");

* First, we create the **root** node with the word "Root".
* Then we attach two children:
  + The left child → "Left Child"
  + The right child → "Right Child"

Now the structure looks like:

Root

/ \

Left Child Right Child

**3. Printing the Values**

Console.WriteLine("Root: " + root.data);

Console.WriteLine("Left: " + root.left.data);

Console.WriteLine("Right: " + root.right.data);

* When we ask for the root’s data, it prints: Root.
* When we ask for the left child’s data, it prints: Left Child.
* When we ask for the right child’s data, it prints: Right Child.

**Program Output:**

Root: Root

Left: Left Child

Right: Right Child

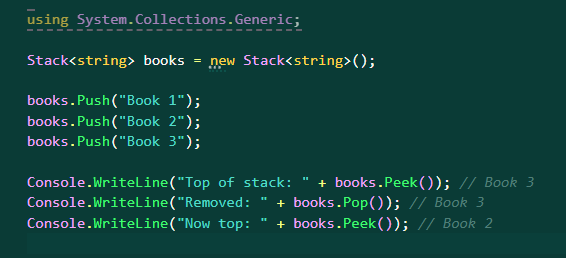
Efficiency (Big O).

* Accessing the root is **instant** → O(1).
* Accessing a child requires moving from the root down one level.
* As trees grow, reaching deeper nodes means more steps.
* In the worst case, finding something in a tree can take **O(n)** (if you check every node).

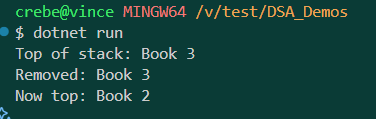
**3. Abstract Data Types (ADT)**

An **Abstract Data Type (ADT)** defines *what* operations are possible, not *how* they’re implemented.

*Code Walkthrough: Stack ADT in C#*



Output:



So what happened:

**1. Creating a Stack**

Stack<string> books = new Stack<string>();

* A **stack** is like a pile of books.
* You can only **add** or **remove** from the **top**.
* Here, we make a new stack that stores strings (book names).

**2. Adding Books (Push)**

books.Push("Book 1");

books.Push("Book 2");

books.Push("Book 3");

* Push means *place a new item on top*.
* After these steps, the stack looks like this (top at the right):
* Bottom → [Book 1, Book 2, Book 3] ← Top
* "Book 3" is now at the top of the stack.

**3. Checking the Top (Peek)**

Console.WriteLine("Top of stack: " + books.Peek()); // Book 3

* Peek means *look at the top without removing it*.
* The top is "Book 3".
* Stack stays the same:
* [Book 1, Book 2, Book 3]

**4. Removing the Top (Pop)**

Console.WriteLine("Removed: " + books.Pop()); // Book 3

* Pop means *remove the top item*.
* "Book 3" is taken off.
* The stack is now:
* [Book 1, Book 2]

**5. Checking the New Top**

Console.WriteLine("Now top: " + books.Peek()); // Book 2

* The new top is "Book 2".
* The stack didn’t lose anything else — it still has Book 1 and Book 2.

**Efficiency (Big O).**

* Pushing a book on top is instant → **O(1)**.
* Peeking at the top is instant → **O(1)**.
* Popping (removing the top) is instant → **O(1)**.

All main operations in a stack take **constant time**, no matter how many items are inside.

**4. Big O Notation**

**Big O notation is a way to measure how efficient an algorithm is.  
It doesn’t measure the exact number of steps, but it shows how the number of steps grows as the input gets bigger.**

**Common Big O Cases**

1. **O(1) → Constant Time**
   * **No matter how big the input is, the time stays the same.**
   * **Example: Getting the first student in a list.**
   * **Even if there are 5 or 5,000 students, you always jump straight to the first one.**
2. **O(n) → Linear Time**
   * **The time grows in a straight line with the input size.**
   * **Example: Reading every student’s name in a class list.**
   * **5 students = 5 steps.**
   * **1,000 students = 1,000 steps.**
3. **O(n²) → Quadratic Time**
   * **The time grows much faster — like a square of the input.**
   * **Example: If every student shakes hands with every other student.**
   * **5 students = 25 handshakes.**
   * **100 students = 10,000 handshakes.**
4. **O(log n) → Logarithmic Time**
   * **The time grows slowly compared to the input size.**
   * **Example: Searching for a word in a dictionary.**
     + **You don’t read every word.**
     + **You open near the middle, check, and cut the search in half.**
   * **1,000 items might only take about 10 steps.**

**Why is this important?**

**Big O helps us predict if a program will still run fast when:**

* **Data grows from small to very large.**
* **Example:**
  + **An algorithm that takes 1 second for 100 items (O(n²))**
  + **Might take 100 seconds for 1,000 items.**
* **But an algorithm that takes 1 second for 100 items (O(n))**
  + **Will only take about 10 seconds for 1,000 items.**

**Learning Tasks**

1. Create an array of 5 integers and print the first element. *(Identify: O(1))*
2. Modify the program to calculate the sum of elements. *(Identify: O(n))*
3. Write a nested loop that prints all element pairs. *(Identify: O(n²))*
4. Discuss: Why does efficiency matter in large datasets like Facebook or banking?

**Practice Problems**

1. Define a data structure and explain why it is important.
2. Differentiate between **linear** and **non-linear** structures.
3. Give two examples of **ADTs**.
4. What is the Big O complexity of accessing the last element of an array?
5. True/False: Big O measures the *exact runtime*.
6. Predict the output:
7. int[] nums = { 5, 10, 15 };
8. Console.WriteLine(nums[2]);

**Lab Assignment**

**Mini-Project:**  
Write a C# program that:

1. Creates an array of 10 integers.
2. Prints the first and last elements (O(1)).
3. Calculates and prints the sum of all elements (O(n)).

**Summary**

* Data structures organize data for efficiency.
* Linear = sequential; Non-linear = hierarchical or networked.
* ADTs define *operations* without specifying implementation.
* Big O helps analyze algorithm performance.
* Choosing the right structure is key to scalable programs.