DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

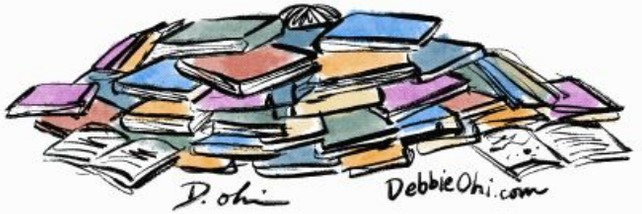
Lecture – 1
Introduction to Data Structures & Algorithms

Instructor: Nasr Kamal

Department of Computer Science and Information Technology

Recommended Books

- “AdamDrozdek__DataStructures_and_Algorithms_in_C_4Ed”
- “A Common-Sense Guide to Data Structures and Algorithms: Level Up Your Core Programming Skills”, Jay Wengrow, Pragmatic Bookshelf, 1st Edition, 2017
- “Data Structures and Algorithms”, Alfred V. Aho, John E. Hopcroft and Jeffrey D.Ullman, Pearson Education Inc., Fourth Impression, 2010



Course Learning Outcomes

- The objective of this course is to introduce the analysis and designing of data structures using various standard algorithms
- Cover well-known data structures such as dynamic arrays, linked lists, stacks, queues, trees and graphs
- Implementation of data structures in C++

Topics Covered

- Introduction to data structures
- Importance of data structures
- Types of data structures
- Space and Time Complexity
- Abstract Data Type (ADT)
- Pseudo Code
- What is an Algorithm
- Importance of Algorithm
- Asymptotic Analysis
- Linear Search
- Binary Search

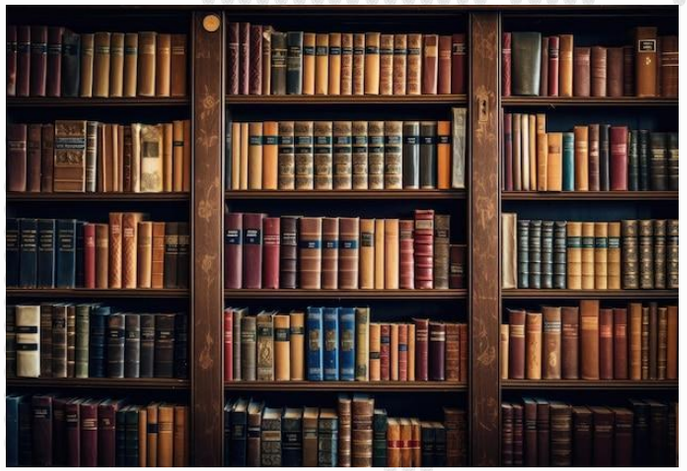
Introduction to Data Structures

- A data structure is a specialized format for organizing, processing, retrieving, and storing data
- It allows a collection of data values to be managed efficiently and provides methods to access and manipulate these values
- Data may be organized in many different ways, the logical or mathematical model of a particular organization of data in memory or on disk is called Data Structure

Importance of Data Structures

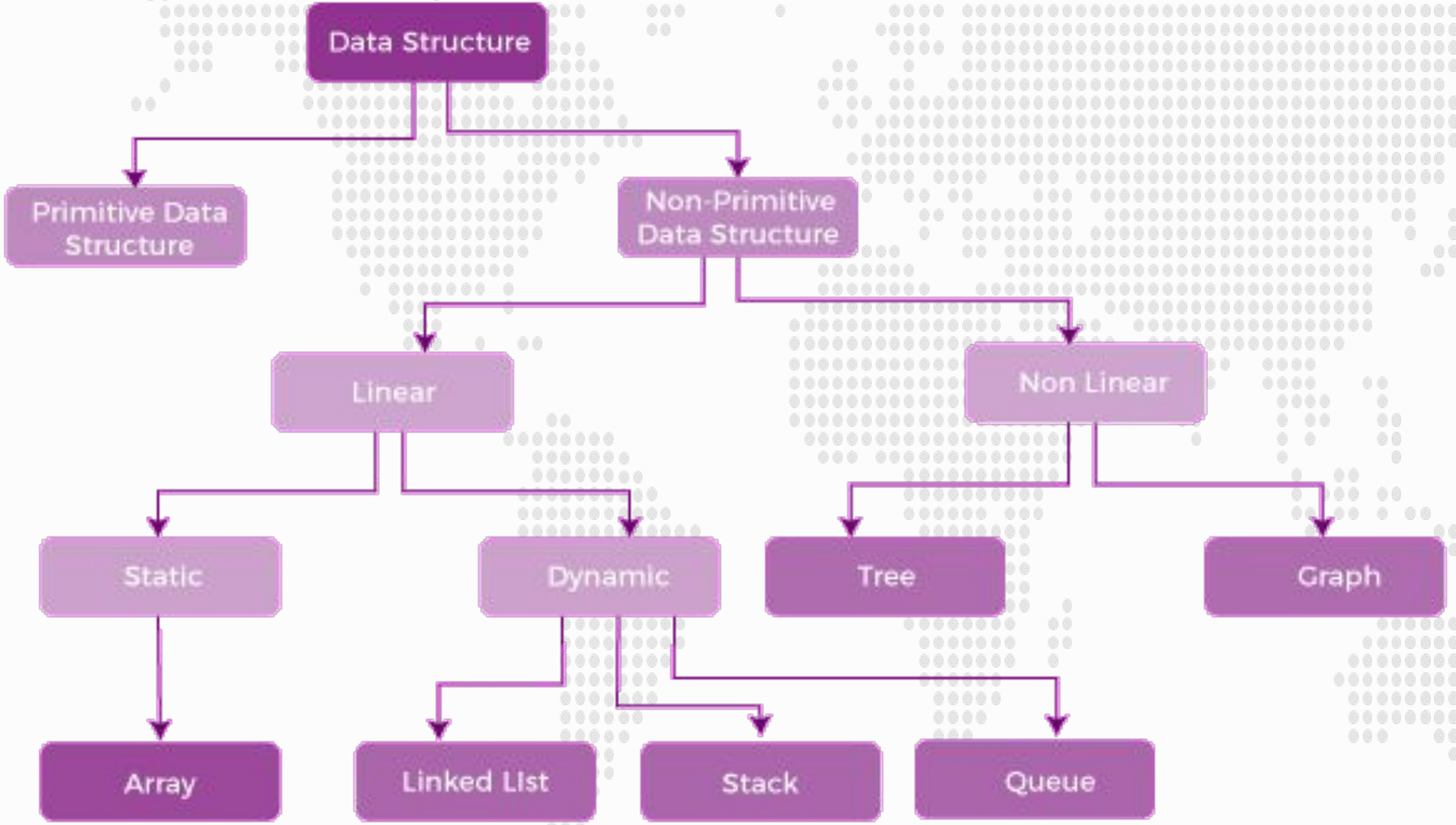
- Data structures are crucial in Computer Science and Software Engineering for designing efficient algorithms and managing large amounts of data

A cluster of books



Books arranged in a shelf

Types of Data Structures



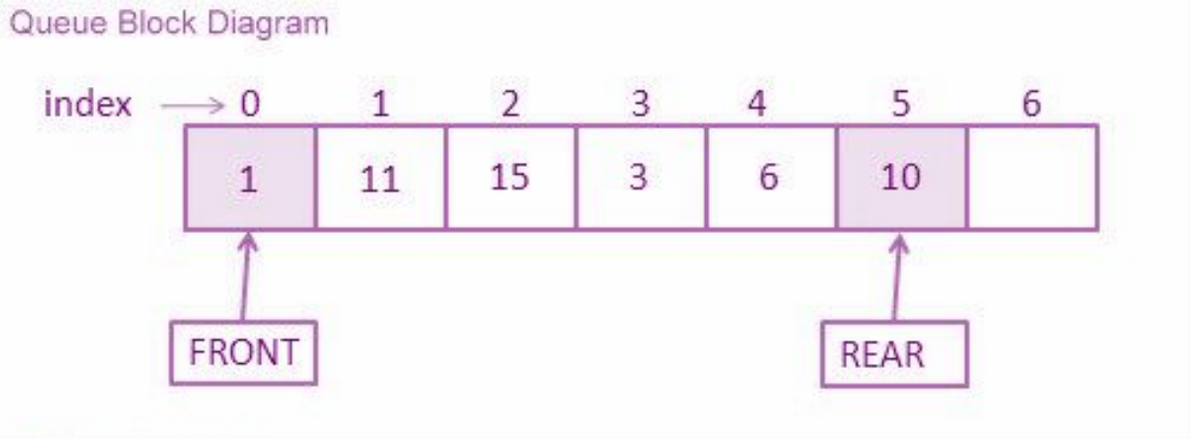
Primitive and Non-Primitive Data Structures

- **Primitive**: Primitive data structures are the basic building blocks of data representation, store simple, atomic values
- Example: Integers, float, string

- **Non-Primitive**: Non-primitive data structures are more complex structures that are derived from primitive data structures
- They can store collections of values and support more complex operations
- Non-primitive data structures can be linear or non-linear and are designed to handle large amounts of data efficiently

Linear Data Structures

- Linear data structures are data structures in which elements are arranged sequentially or linearly, where each element is connected to its previous and next element in a single level, forming a linear sequence
- They allow traversing all elements in a single run and are relatively easy to implement and understand
- Linear data structures are further categorize into static and dynamic data structures



Linear Data Structures Contd.

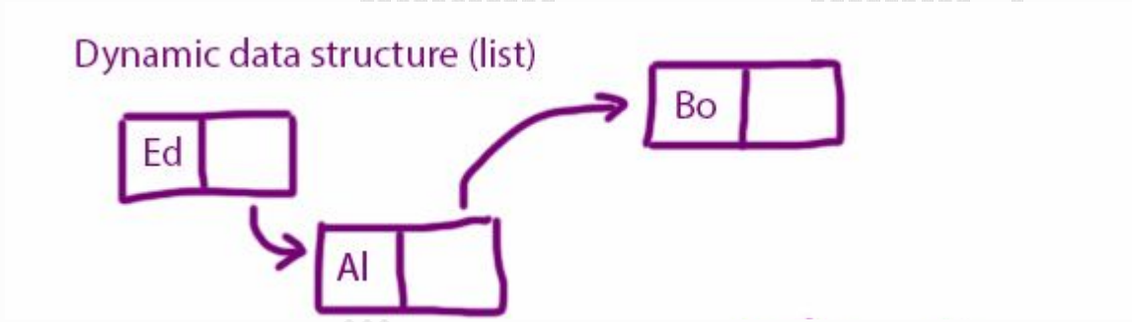
Static Data Structures

- Fixed Size: The size is determined at compile-time and cannot be changed
- Contiguous Memory Allocation: Elements are stored in contiguous memory locations, allowing for efficient access using indices
- Ease of Use: Simple to implement and use due to their fixed size and memory allocation
- Example: Arrays

Linear Data Structures Contd.

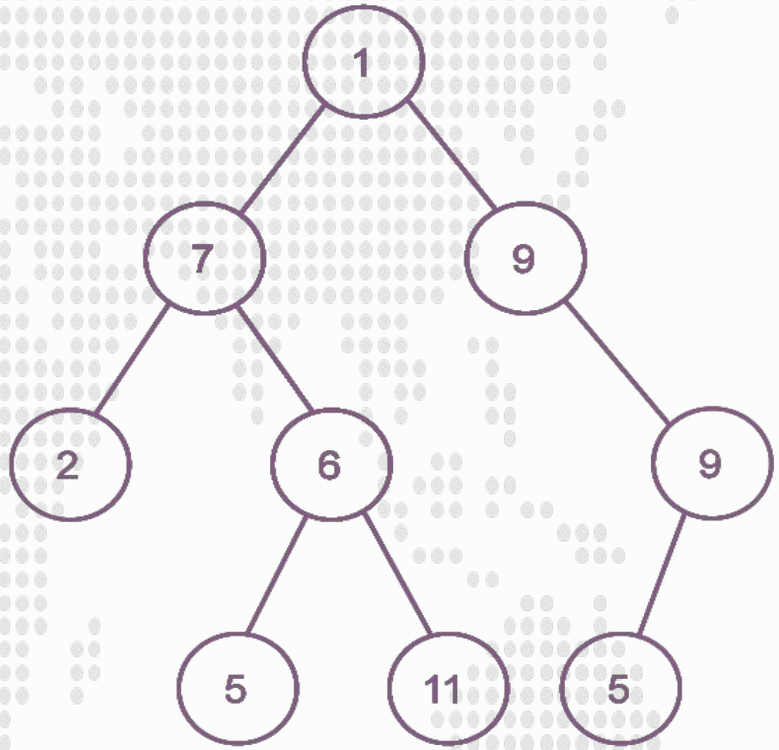
Dynamic Data Structures

- Variable Size: The size can change during runtime based on the number of elements
- Flexible Memory Allocation: Memory is allocated and de-allocated as needed, allowing for efficient use of memory
- Complex Management: More complex to implement and manage due to the need for dynamic memory allocation and de-allocation
- Example: Linked-list, stacks, queues



Non-Linear Data Structures

- Non-linear data structures are data structures where elements are not arranged in a sequential or linear manner
- Instead, they are arranged in a hierarchical or interconnected manner, where each element can be connected to multiple elements, allowing for more complex relationships between elements
- Example: Trees and graphs



Operations Performed on Data Structures

Data structures support various operations that allow for the efficient manipulation and retrieval of data

- **Insertion:** Adding a new element to the data structure
- **Deletion:** Removing an element from the data structure
- **Traversal:** Accessing each element of the data structure in a systematic manner
- **Searching:** Finding a specific element in the data structure

Space and Time Complexity

- **Space Complexity:** refers to the amount of memory a data structure or algorithm uses relative to the size of the input data
- **Notation**: space complexity is often expressed using Big O notation, which describes the upper bound of the space required as a function of the input size n
- **Example**: an array of size n has a space complexity of $O(n)$
- **Time Complexity:** refers to the amount of time a data structure or algorithm takes to run as a function of the size of the input data
- It provides an estimate of the number of basic operations or steps an algorithm performs relative to the input size
- **Notation:** time complexity is often expressed using Big O notation, which describes the upper bound of the time required as a function of the input size n
- **Example:** a simple linear search in an array of size n has a time complexity of O(n) because, in the worst case, it needs to examine each element once

Time Complexity Contd.

- **Components:**
- **Best Case:** The minimum time an algorithm takes to complete, given the most favourable input
- **Worst Case:** The maximum time an algorithm takes to complete, given the least favourable input
- **Average Case:** The expected time an algorithm takes to complete, averaged over all possible inputs

Common Time Complexity Classes:

- Constant Time – O(1): Accessing an element in an array by index
- Logarithmic Time – O(logn): Binary search in a sorted array
- Linear Time – O(n): Traversing all elements in an array

Pseudo-Code

- A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm
- In the context of data structures, pseudocode is often used to explain how various operations (such as insertion, deletion, searching, etc.) are performed on data structures like arrays, linked lists, trees, graphs, etc

Eg: Algorithm array Max(A,n)
Input: An array A storing n integers.
Output: The maximum element in A.

```
currentMax =  A[0]
for i =1 to n-1
    if  currentMax < A[i] then currentMax = A[i]
return currentMax
```

Pseudocode

It is more structured than usual prose but less formal than a programming language.

How to write a pseudocode:

- Start with the Problem Statement: Clearly understand what the algorithm needs to achieve.
- Define Inputs and Outputs: Clearly state what inputs the algorithm will take and what output it should produce.
- Use Control Structures: Utilize loops, conditionals (if, else), and functions to break down the problem.
- Focus on Clarity: Make sure the pseudocode is easy to follow and understand.
- Abstract Complex Operations: Don't get bogged down in implementation details. If a function is complex, you can simply describe what it does without going into too much detail

Pseudocode

Expressions:
Same as Programming
Method Declarations:
Algorithm name(param1, param2)
Programming Constructs:
Decision structures: if…then….[else]
While-loop: while….do
Repeat-loop: repeat……until….
For-loop: for
Array indexing: A[i], A[I,j]
Methods:
Calls: object method(args)
Return: return value

Algorithm

- An algorithm is a step-by-step procedure or a set of rules to be followed in calculations or other problem-solving operations, particularly by a computer
- In the context of data structures, algorithms are used to manipulate and manage data stored in various structures efficiently

Analysis of an Algorithm

Analysing an algorithm involves evaluating its efficiency and correctness. The main aspects of analysis include:

- **Correctness:** Ensures the algorithm provides the correct output for all possible inputs
- **Efficiency:** Algorithms help perform operations like searching, sorting, insertion, deletion, and traversal efficiently, minimizing time and space complexity
- **Optimization:** Good algorithms optimize the use of resources, such as memory and processing power, ensuring that programs run faster and consume less memory
- **Problem Solving:** Algorithms provide systematic approaches to solving problems and implementing solutions in software development
- **Scalability:** Efficient algorithms enable systems to handle large datasets and scale as the amount of data grows

Measuring the running time of an Algorithm

Empirical Analysis:

- Write a program that implements the algorithm
- Run the program with data sets of varying size and composition
- Use a method like startTime and stopTime to get an accurate measure of the actual running time (absolute speed)

Limitations:

- It is necessary to implement and test the algorithm in order to determine its running time
- Experiments can be done only a limited set of inputs, and may not be indicative of the running time on other inputs not included in the experiment
- In order to compare two algorithms, the same hardware and software environments should be used

Measuring the running time of an Algorithm

Theoretical Analysis:

Asymptotic Analysis: It provides a way to compare algorithms independently of the specific hardware or implementation details. For large input sizes, relative speed (asymptotic complexity) becomes crucial to ensure the algorithm can handle the data efficiently

- Big O Notation (O): Describes the upper bound of the running time. It provides the worst-case scenario
- Big Ω Notation (Ω): Describes the lower bound of the running time. It provides the best-case scenario
- Big Θ Notation (Θ): Describes the exact bound of the running time, encompassing both the upper and lower bounds

Measuring the running time of an Algorithm

Steps for Theoretical Analysis:

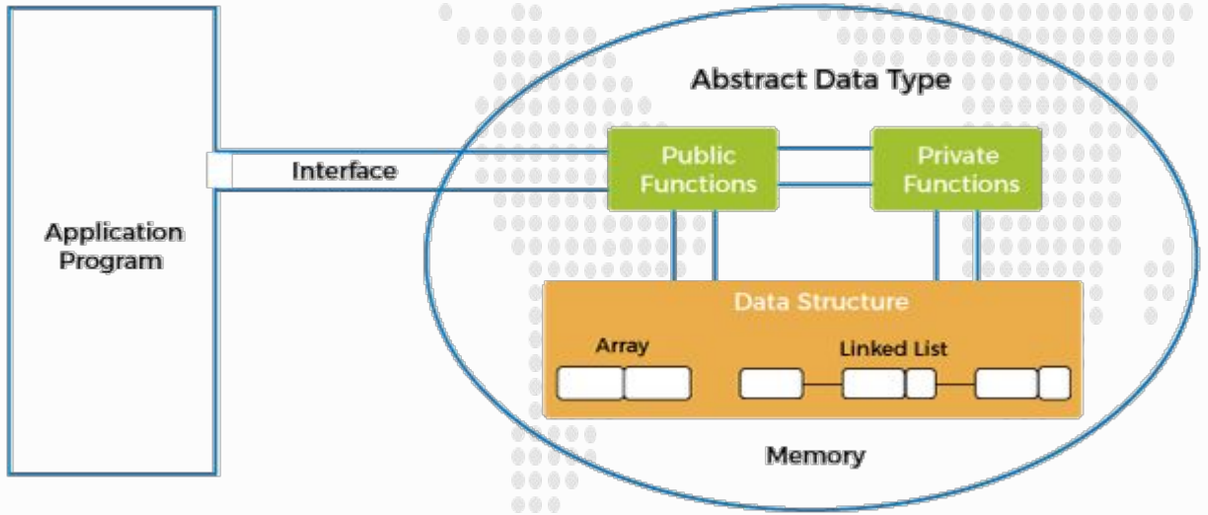
- Identify Basic Operations: Determine the fundamental operations of the algorithm, such as comparisons, assignments, or arithmetic operations
- Count Operations: Calculate the number of times each basic operation is executed as a function of the input size n
- Determine Dominant Term: Identify the term that grows the fastest as n increases. This term determines the algorithm's asymptotic complexity
- Express Complexity: Use Big O, Big Ω, or Big Θ notation to express the running time

Time-Space Tradeoff

- In computer science, a space–time or time–memory tradeoff is a way of solving a problem or calculation in less time by using more storage space (or memory), or by solving a problem in very little space by spending a long time
- So if your problem is taking a long time but not much memory, a space–time tradeoff would let you use more memory and solve the problem more quickly
- Or, if it could be solved very quickly but requires more memory than, you can try to spend more time solving the problem in the limited memory

Abstract Data Type (ADT)

- Abstract Data Types (ADTs) are models for data structures that define the type of data stored, the operations allowed on them, and the types of parameters of the operations
- ADTs provide a high-level way to describe the behaviour of data structures without specifying implementation details.



Example Array ADT

- An Array ADT is a collection of elements identified by index or key, where each element is stored in a contiguous block of memory. The operations on an Array ADT are defined abstractly, without specifying the implementation details
- **Operations on Array ADT:**
- Accessing Elements (Get): Retrieve the value of an element at a specific index
- Inserting Elements: Insert an element at a specific position
- Deleting Elements: Remove an element from a specific position
- Updating Elements (Set): Modify the value of an element at a specific index
- Searching: Find an element with a specific value
- Traversal: Access each element in the array, typically from start to end

Example Array ADT

- The Array ADT provides a clear and structured way to define and interact with arrays. By abstracting away the implementation details, it allows users to focus on the operations and how they interact with the data, making the code more modular and easier to manage

Linear Search

- The Array ADT provides a clear and structured way to define and interact with arrays. By abstracting away the implementation details, it allows users to focus on the operations and how they interact with the data, making the code more modular and easier to manage
- Linear Search is the simplest searching algorithm. It works by checking each element in a list or array one by one until the target value is found or the list ends
- Time Complexity: O(n) where n is the number of elements in the array.
- Best Case: The element is the first one in the array, O(1).
- Worst Case: The element is the last one or not in the array at all, O(n).

Linear Search

How Linear Search Works:

- Start from the first element of the array.
- Compare the target value with the current element.
- If the current element matches the target, return its index.
- If not, move to the next element and repeat the process.
- If the target is not found by the end of the array, return –1 indicating the target is not present.

Linear Search

Suppose you have the following array and want to find the number 16:

Array: [3, 8, 12, 5, 16, 23] Target: 16

- Start at index 0: 3 (not a match)
- Move to index 1: 8 (not a match)
- Move to index 2: 12 (not a match)
- Move to index 3: 5 (not a match)
- Move to index 4: 16 (match found!)

The algorithm returns the index 4

```
int linearSearch(int arr[], int n, int
target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i; // Target found
        }
    }
    return -1; // Target not found
}
```

Binary Search

Binary Search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half

- Time Complexity: O(log n) where n is the number of elements in the array.
- Best Case: The element is the middle one in the array, O(1).
- Worst Case: The element is not in the array, O(log n).

Binary Search

How Binary Search Works:

- Begin with the middle element of the sorted array.
- Compare it with the target value.
- If it matches, return the index.
- If the target is smaller, narrow the search to the left half.
- If the target is larger, narrow the search to the right half.
- Repeat the process until the target is found or the search interval is empty.

Binary Search

Suppose you have the following sorted array and want to find the number 16:

Array: [3, 8, 12, 16, 23, 30] Target: 16

- Start with the middle element: 12 (not a match, but 16 > 12 so move to the right half)
- Now consider the right half: [16, 23, 30]
- Middle element in the new half is 23 (not a match, but 16 < 23, so move to the left)
- Now consider the left half: [16]
- The middle element is 16 (match found!)

The algorithm returns the index 3.

Binary Search

```
int binarySearch(int arr[], int n, int
target)
{
 int left = 0; int right = n - 1;
 while (left <= right) {
int mid = left + (right - left) / 2;
// Check if the target is at mid
if (arr[mid] == target) {
return mid; // Target found }
// If target is greater, ignore left half
if (arr[mid] < target) {
 left = mid + 1;
}
// If target is smaller, ignore right
half
else { right = mid - 1; }
 }
 return -1;
 // Target not found }
```

Comparison of Linear and Binary Search

Use Cases:

- Linear Search: Can be used on unsorted or sorted arrays. It's simple but less efficient for large arrays.
- Binary Search: Requires a sorted array. It's much faster for large arrays, with a time complexity of O(log n).

Efficiency:

- Linear Search: O(n), where you might have to check every element.
- Binary Search: O(log n), where the search space is halved each time.

Ease of Implementation:

- Linear Search: Easy to implement and understand.
- Binary Search: Slightly more complex due to the need for array sorting and handling of mid-points.

Task for Students

1. Given an array int arr[] = {3, 8, 12, 5, 16, 23} and target is 5, implement a linear search to find a target element using while and do while loop

2. Given a sorted array int arr[]={10,20,38,46,52,64}, implement binary search to find a target element 52