



# Aror University of Art, Architecture, Design & Heritage Sukkur

Department of AI-Multimedia and Gaming

---

Week 01: Fundamentals of array data structure

Date: August 19, 2024

Subject: Data Structure (CSC221), Fall 2024

Instructor: Abdul Ghafoor

---

## Introduction to the Course

Data Structures are the fundamental building blocks in computer science, enabling efficient storage, retrieval, and manipulation of data. This course will delve into the implementation and application of various data structures using the Java programming language. By the end of this course, you will have a strong understanding of how data structures work and how to use them to solve complex problems efficiently.

### Course Objectives

The primary objectives of this course are as follows:

**Understanding Fundamental Data Structures:** Data structures like arrays, linked lists, stacks, queues, trees, and graphs are crucial for organizing data efficiently. This course will provide a deep dive into these structures, explaining their properties, usage scenarios, and operations.

**Implementation and Manipulation in Java:** You will learn how to implement these data structures in Java, a versatile and widely-used programming language. The course will cover both the theory behind these structures and practical coding examples.

**Problem-Solving Skills:** Through assignments and labs, you will develop problem-solving skills by applying appropriate data structures to various real-world scenarios. This hands-on approach ensures that you can not only understand but also apply the concepts learned.

**Performance Analysis:** Evaluating the efficiency of data structures is critical. You will learn to analyze the time and space complexity of different data structures and algorithms, helping you make informed decisions when optimizing code.

### Course Structure

The course is divided into the following components:

**Lectures:** Each lecture will introduce new concepts and provide detailed explanations, often with examples and code snippets.

**Assignments:** Assignments are designed to reinforce the concepts learned in lectures through practical coding exercises.

**Labs:** In labs, you will implement data structures under supervision, allowing you to practice and ask questions in real-time.

**Exams/Quizzes:** Regular assessments will test your understanding of the material and your ability to apply it.

### Tools and IDEs

For this course, we will be using the following tools and Integrated Development Environments (IDEs):

- **Java Development Kit (JDK):** Ensure you have the latest version of JDK installed on your system.
- **Eclipse/IntelliJ Netbeans:** These are recommended IDEs for Java development. They offer powerful features that will assist you in coding, debugging, and testing your data structure implementations.
- **Version Control (Git/GitHub):** Familiarize yourself with version control systems like Git. You will be required to submit assignments through GitHub, which also helps you track your progress and collaborate with others.

### Importance of Data Structures

Data structures are crucial for any computer science professional. They allow for the efficient organization and manipulation of data, which is essential for developing high-performance software. For example, consider the task of searching for an element in a dataset. If the dataset is organized in a simple array, the search operation might take linear time ( $O(n)$ ). However, if the data is organized using a binary search tree, the search operation can be optimized to logarithmic time ( $O(\log n)$ ), significantly improving performance.

## What is a Data Structure?

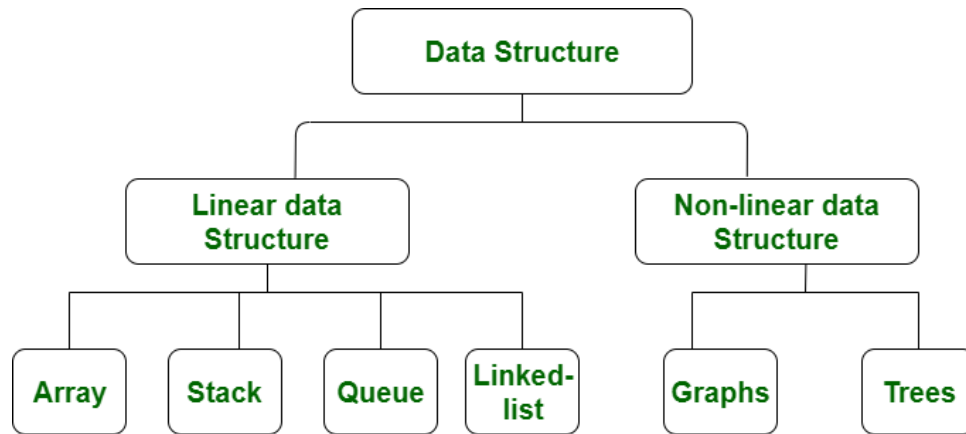
Data structures are specialized formats for organizing, processing, retrieving, and storing data. They provide a way to manage large amounts of data efficiently, which is crucial for tasks such as search, insertion, deletion, and updating.

### Definition and Concept

A data structure is a systematic way of organizing data to perform operations efficiently. Examples include arrays, linked lists, stacks, queues, trees, and graphs.

### Types of Data Structures

Linear and Non-Linear Data Structures



### Linear Data Structures

Linear data structures arrange data elements sequentially, one after the other. Each element in a linear data structure is connected to its previous and next element, forming a linear order. These structures are easy to implement and manage, making them fundamental to many algorithms and applications.

Examples of Linear Data Structures: Arrays, Linked List, Stack, Queue

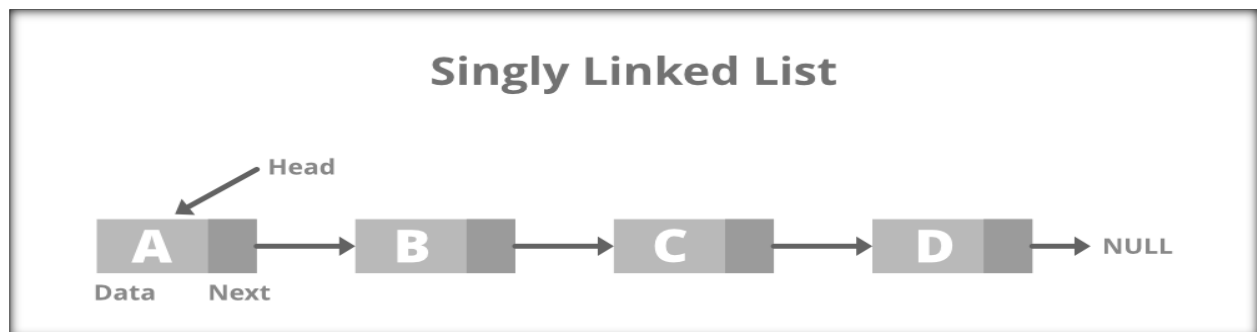
#### 1. Arrays:

A fixed-size, linear collection of elements stored in contiguous memory locations. Each element can be accessed directly by its index.

```
int[] nums = {10, 20, 30, 40, 50};
```

#### 2. Linked Lists:

A sequence of elements where each element (node) contains data and a reference (or link) to the next node in the sequence. Unlike arrays, linked lists can easily grow and shrink in size.



#### 3. Stacks:

A linear data structure that follows the Last In, First Out (LIFO) principle. Elements can only be added or removed from the top of the stack.

Example: A stack of books where you can only take the top book or add a new book on top.



#### 4. Queues:

A linear data structure that follows the First In, First Out (FIFO) principle. Elements are added from the rear and removed from the front.

Example: A line of people waiting for a service where the person who arrives first is served first.



## Non-Linear Data Structures

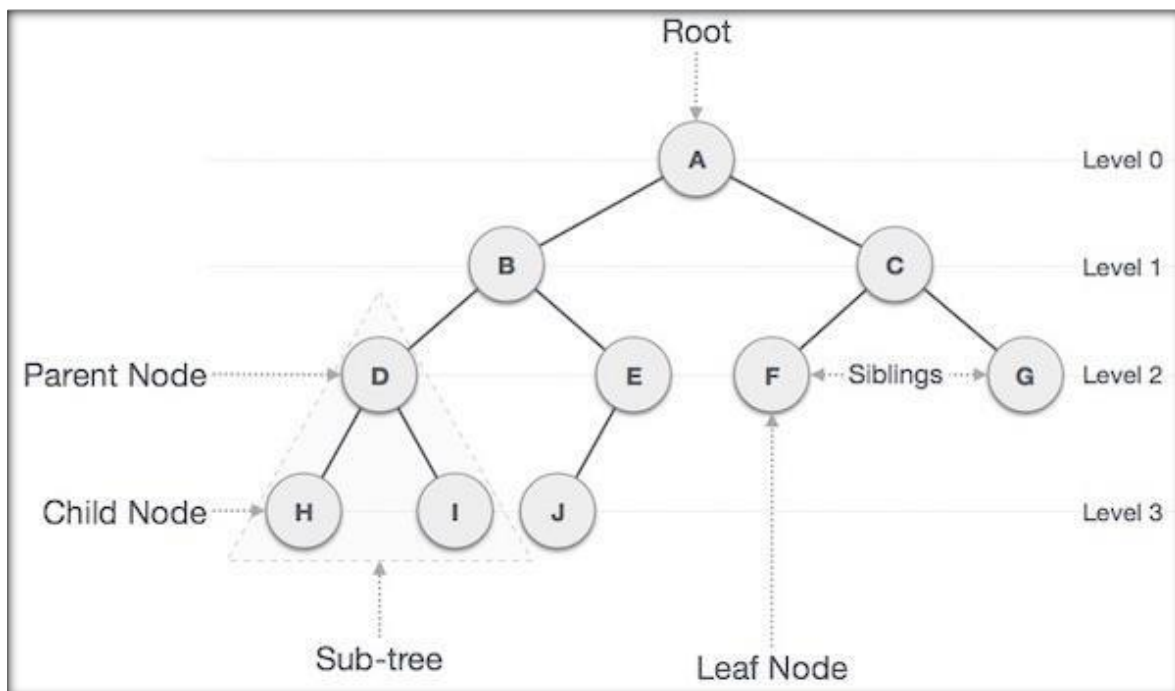
Non-linear data structures do not arrange elements sequentially. Instead, they organize data in a hierarchical or interconnected manner. This structure allows for more complex relationships between elements, making them suitable for advanced algorithms and data representation.

Examples of Non-Linear Data Structures: Tree, and Graphs

### 1. Trees:

A hierarchical data structure consisting of nodes connected by edges. The topmost node is called the root, and each node can have zero or more child nodes. Common types include binary trees, AVL trees, and binary search trees.

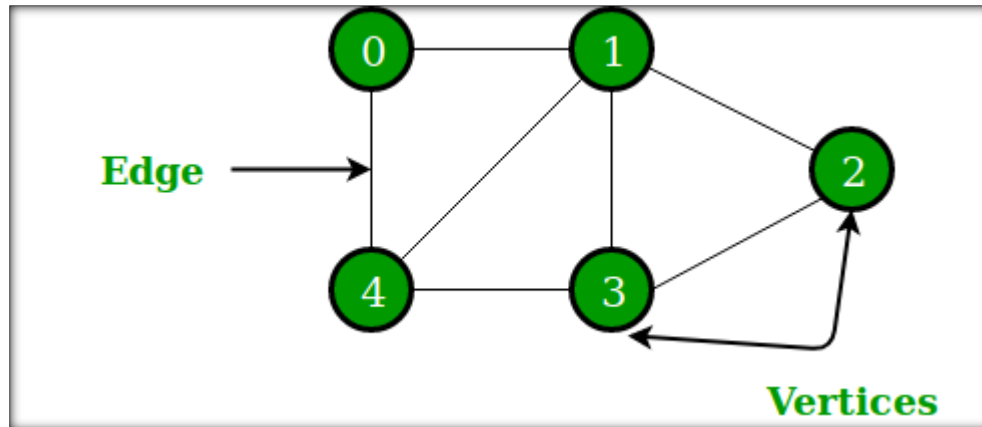
Example: A family tree, where each person (node) can have multiple children but only one parent.



### 2. Graphs:

A network of nodes (also called vertices) connected by edges. Graphs can be directed or undirected and are used to represent complex relationships such as social networks, transportation networks, and dependency graphs.

Example: A map where cities are nodes and roads connecting them are edges.



## Need for Data Structures

### Efficient Data Management

Data structures are essential for organizing and managing data efficiently. The choice of data structure can greatly affect the performance of an algorithm, particularly in terms of time complexity and space complexity.

**Example:** Consider searching for an element in a dataset. If the dataset is stored in an unsorted array, a linear search would take  $O(n)$  time. However, if the dataset is stored in a binary search tree, the search operation can be optimized to  $O(\log n)$  time, significantly improving performance.

### **Optimized Algorithms**

Data structures play a crucial role in optimizing algorithms by providing the right tools to manage and access data efficiently. When you choose an appropriate data structure for a specific algorithm, it can significantly improve the algorithm's performance and correctness.

**Example:** Breadth-First Search (BFS) is an algorithm used to traverse or search through all the nodes in a graph or tree. BFS explores all the nodes at the present depth level before moving on to nodes at the next depth level. To achieve this, BFS needs a data structure that can keep track of the nodes to be explored in the correct order.

## What is an Array?

An array is a collection of elements, all of the same type, stored in contiguous memory locations. Each element can be accessed directly by its index, making arrays a fundamental and efficient way to store and manage data.



The diagram illustrates an array as a contiguous block of memory. It consists of a table with three rows and five columns. The first row represents memory addresses, starting from 2391 and increasing by 1 up to 2395. The second row represents the values stored at these addresses: 12, 34, 68, 77, and 43. The third row represents the array indices, starting from 0 and increasing by 1 up to 4. Arrows point from the labels 'Memory Address', 'Array Values', and 'Array Index' to their respective rows in the table.

|                |      |      |      |      |      |
|----------------|------|------|------|------|------|
| Memory Address | 2391 | 2392 | 2393 | 2394 | 2395 |
| Array Values   | 12   | 34   | 68   | 77   | 43   |
| Array Index    | 0    | 1    | 2    | 3    | 4    |

- **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.

### Characteristics of Arrays:

- **Fixed Size:** Once an array is created, its size cannot be changed. This means you must specify the number of elements (the array's size) when you create it.
- **Homogeneous:** All elements in an array must be of the same type, whether they are int, char, String, or any other data type.
- **Index-Based Access:** Elements in an array can be accessed quickly using their index. The index of the first element is always 0, and the last element's index is `array.length - 1`.
- **Memory Contiguity:** Elements are stored in contiguous memory locations, which allows for fast access and efficient use of memory.

## Creating Array in Java

### Declaration

```
package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {
        // Array declaration
        int[] numbers; // Declares an array of integers
        String[] names; // Declares an array of Strings
    }
}
```

### Initialization

```
package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {
        // Array declaration
        int[] numbers; // Declares an array of integers
        String[] names; // Declares an array of Strings

        // Array initialization
        numbers = new int[5]; // Creates an array of 5 integers
        names = new String[5]; // Creates an array of 5 Strings
    }
}
```

### Array declaration and Initialization

```
package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {
        // Array declaration and initialization together
        int[] numbers = new int[5]; // Creates an array of 5 integers
    }
}
```

### Array declaration and Initialization with values

```

package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {
        // Array declaration and initialization together
        int[] numbers = {1,2,4,7,11}; // Declares, creates, and initializes with values
    }
}

```

## Basic Operations in Arrays

The basic operations in the Arrays are insertion, deletion, searching, display, traverse, and update. These operations are usually performed to either modify the data in the array or to report the status of the array.

Following are the basic operations supported by an array.

- Traverse – print all the array elements one by one.
- Insertion – Adds an element at the given index.
- Deletion – Deletes an element at the given index.
- Search – Searches an element using the given index or by the value.
- Update – Updates an element at the given index.
- Display – Displays the contents of the array.

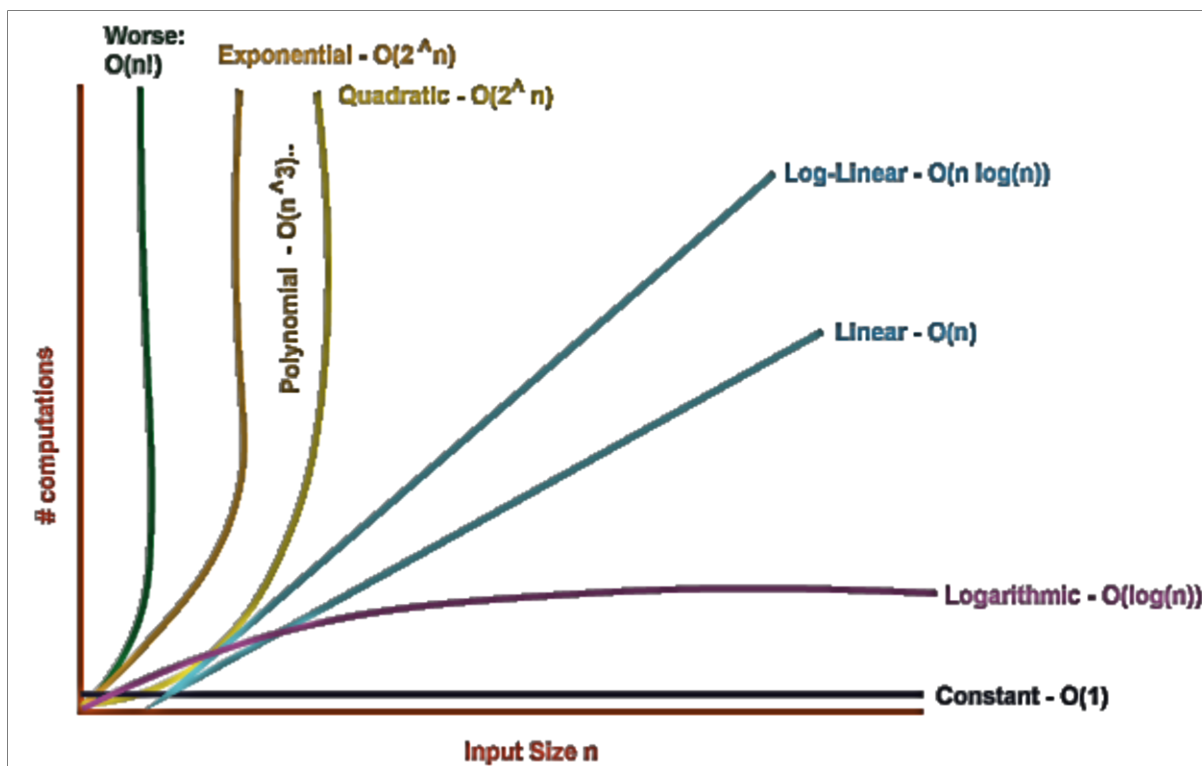
| Operation         | Time Complexity | Explanation   |
|-------------------|-----------------|---|
| Traverse          | $O(n)$          | Visits each element in the array.                                       |
| Insertion         | $O(n) / O(1)$   | $O(n)$ if inserting at the beginning/middle; $O(1)$ if at the end.      |
| Deletion          | $O(n)$          | Shifts elements after deletion.   |
| Search (By Index) | $O(1)$          | Direct access to the element using its index.                           |
| Search (Linear)   | $O(n)$          | Checks each element until the value is found.                           |
| Search (Binary)   | $O(\log n)$     | Assumes the array is sorted; divides search space in half at each step. |
| Update            | $O(1)$          | Directly accesses and modifies the element.                             |
| Display           | $O(n)$          | Visits each element to display it.                                      |

## Time complexity

Time complexity is a measure of the amount of time an algorithm takes to run as a function of the size of the input. It helps in evaluating the efficiency of an algorithm, especially in terms of how it scales with increasing input size.

Here's a list of time complexities from best (fastest) to worst (slowest):

| Name             | Time Complexity |
|------------------|-----------------|
| Constant Time    | $O(1)$          |
| Logarithmic Time | $O(\log n)$     |
| Linear Time      | $O(n)$          |
| Quasilinear Time | $O(n \log n)$   |
| Quadratic Time   | $O(n^2)$        |
| Exponential Time | $O(2^n)$        |
| Factorial Time   | $O(n!)$         |



### Length of array

In Java, you can find the length of an array using the `.length` property. This property returns the number of elements in the array.

```

package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {
        // Array declaration and initialization
        int[] numbers = new int[5]; // Creates an array of 5 integers
        int total_length = numbers.length;

        System.out.println(total_length); // output is 5
    }
}

```

## Insertion Operation

First, let's declare and initialize an array of 5 integers. Then, we'll print all the elements of the array to check the default values stored in it.

```

package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {
        // Array declaration and initialization
        int[] numbers = new int[5]; // Creates an array of 5 integers
        // Printing all elements to check the default values
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }
    }
}

```

```

--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Element at index 0: 0
Element at index 1: 0
Element at index 2: 0
Element at index 3: 0
Element at index 4: 0

-----
BUILD SUCCESS
-----

Total time: 1.565 s
Finished at: 2024-08-14T18:06:55+05:00
-----

```

In Java, when an array of integers is initialized, it is automatically filled with the default value of 0 for each element.

Here's a list of default values for different data types in Java:

Primitive Data Types:

- int: 0
- short: 0
- long: 0L
- float: 0.0f
- double: 0.0d
- char: '\u0000' (null character)
- boolean: false

## Insertion at the Start of an Array in Java

Inserting a value at the start of an array requires shifting all existing elements one position to the right to make space for the new element at index 0.

```
package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {
        // Step 1: Declare and initialize an array
        int[] arr = new int[6]; // An array with space for 6 elements
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        arr[3] = 40;
        arr[4] = 50;

        // Step 2: Print the array before insertion
        System.out.println("Array before insertion:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }

        // Step 3: Insert a new element (5) at the start (index 0)
        int newElement = 5;
        // Shift elements to the right to make space at index 0
        for (int i = arr.length - 1; i > 0; i--) {
            arr[i] = arr[i - 1];
        }
        // Insert the new element at the start
        arr[0] = newElement;

        // Step 4: Print the array after insertion
        System.out.println("\nArray after insertion at the start:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }
    }
}
```

```
--- compiler:3.11.0:compile (default-compile) @ DataStructure ---
Changes detected - recompiling the module! :source
Compiling 1 source file with javac [debug target 21] to target\c

--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Array before insertion:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
Element at index 5: 0

Array after insertion at the start:
Element at index 0: 5
Element at index 1: 10
Element at index 2: 20
Element at index 3: 30
Element at index 4: 40
Element at index 5: 50
-----
BUILD SUCCESS
-----
Total time: 1.742 s
Finished at: 2024-08-17T16:00:08+05:00
-----
```

## Insertion at the Middle of an Array in Java

Inserting a value in the middle of an array requires shifting all elements after the insertion point one position to the right to create space for the new element.

```
public class DataStructure {

    public static void main(String[] args) {
        // Step 1: Declare and initialize an array
        int[] arr = new int[6]; // An array with space for 6 elements
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        arr[3] = 40;
        arr[4] = 50;

        // Step 2: Print the array before insertion
        System.out.println("Array before insertion:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }

        // Step 3: Insert a new element (25) in the middle (at index 2)
        int newElement = 25;
        int position = 2; // Middle position

        // Shift elements to the right to make space for the new element
        for (int i = arr.length - 1; i > position; i--) {
            arr[i] = arr[i - 1];
        }

        // Insert the new element at the desired position
        arr[position] = newElement;

        // Step 4: Print the array after insertion
        System.out.println("\nArray after insertion in the middle:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }
    }
}
```

```
--- compiler:3.11.0:compile (default-compile) @ DataStructure ---
Changes detected - recompiling the module! :source
Compiling 1 source file with javac [debug target 21] to target\cl

--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Array before insertion:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
Element at index 5: 0

Array after insertion in the middle:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 25
Element at index 3: 30
Element at index 4: 40
Element at index 5: 50

-----
BUILD SUCCESS
-----

Total time: 1.744 s
Finished at: 2024-08-17T16:18:48+05:00
-----
```

## Insertion at the End of an Array in Java

Inserting a value at the end of an array is the simplest insertion operation. If the array has enough space, you can directly add the new element at the last available index without needing to shift any existing elements.

```
package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {

        // Step 1: Declare and initialize an array
        int[] arr = new int[6]; // An array with space for 6 elements
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        arr[3] = 40;
        arr[4] = 50;

        // Step 2: Print the array before insertion
        System.out.println("Array before insertion:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }

        // Step 3: Insert a new element (60) at the end
        int newElement = 60;

        // Insert the new element at the last available position
        arr[arr.length - 1] = newElement;

        // Step 4: Print the array after insertion
        System.out.println("\nArray after insertion at the end:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }
    }
}
```

```
--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Array before insertion:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
Element at index 5: 0

Array after insertion at the end:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
Element at index 5: 60

-----
BUILD SUCCESS
-----

Total time: 1.315 s
Finished at: 2024-08-18T09:35:14+05:00
-----
```

## Traversal Operation

Traversal in an array involves visiting each element of the array sequentially. This operation is typically used to access, print, or modify each element.

```
package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {

        int[] arr = {10, 20, 30, 40, 50};

        // Traverse and print each element of the array
        System.out.println("Array elements:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }
    }
}
```

```
--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Array elements:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50

-----
BUILD SUCCESS
-----

Total time: 1.264 s
Finished at: 2024-08-18T09:42:27+05:00
-----
```

## Deletion Operation

Deletion in an array involves removing an element from a specific index. Since arrays have a fixed size, after deletion, the elements following the deleted element need to be shifted one position to the left to fill the gap.

```
package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {

        int[] arr = {10, 20, 30, 40, 50};

        // Deleting the element at index 2 (element 30)
        int deleteIndex = 2;

        // Shift elements to the left to fill the gap
        for (int i = deleteIndex; i < arr.length - 1; i++) {
            arr[i] = arr[i + 1];
        }

        // Optional: Set the last element to a default value or ignore it
        arr[arr.length - 1] = 0; // Assuming 0 as a default value for int array

        // Traverse and print the array after deletion
        System.out.println("Array after deletion:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }
    }
}
```

```

--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Array after deletion:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 40
Element at index 3: 50
Element at index 4: 0

-----

BUILD SUCCESS

-----

Total time: 1.290 s
Finished at: 2024-08-18T09:47:40+05:00
-----

```

## Searching in an Array in Java

Searching in an array can be done in several ways, depending on the specific needs and whether the array is sorted or not. Below, we'll cover three common search methods: search by index, linear search, and binary search.

### 1. Search by Index

Searching by index is the most straightforward way to access an element in an array. Since arrays in Java are indexed, you can directly access any element using its index.

```

package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {

        int[] arr = {10, 20, 30, 40, 50};

        // Access element at index 2
        int index = 2;
        int element = arr[index];

        System.out.println("Element at index " + index + ": " + element);

    }

}

```

```

--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Element at index 2: 30

-----

BUILD SUCCESS

-----

Total time: 1.521 s
Finished at: 2024-08-18T09:52:55+05:00
-----

```

## 2. Linear Search

Linear search involves scanning the array from the beginning to the end to find a specific value. This method does not require the array to be sorted.

```
package com.mycompany.datastructure;

public class DataStructure {

    public static void main(String[] args) {

        int[] arr = {10, 20, 30, 40, 50};
        int target = 30;
        boolean found = false;

        // Linear search
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                System.out.println("Element " + target + " found at index " + i);
                found = true;
                break;
            }
        }

        if (found == false) {
            System.out.println("Element " + target + " not found in the array.");
        }
    }
}
```

```
--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Element 30 found at index 2
-----
BUILD SUCCESS
-----
Total time: 1.299 s
Finished at: 2024-08-18T09:56:26+05:00
-----
```

### 3. Binary Search

Binary search is a more efficient method that requires the array to be sorted. It works by repeatedly dividing the search interval in half. If the target value is less than the middle element, the search continues in the lower half; otherwise, it continues in the upper half.

```
package com.mycompany.datastructure;
import java.util.Arrays;
public class DataStructure {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int target = 30;
        // Ensure the array is sorted (binary search requires this)
        Arrays.sort(arr);

        int left = 0;
        int right = arr.length - 1;
        int middle;
        boolean found = false;

        // Binary search
        while (left <= right) {
            middle = left + (right - left) / 2;

            if (arr[middle] == target) {
                System.out.println("Element " + target + " found at index " + middle);
                found = true;
                break;
            }
            if (arr[middle] < target) {
                left = middle + 1;
            } else {
                right = middle - 1;
            }
        }
        if (found == false) {
            System.out.println("Element " + target + " not found in the array.");
        }
    }
}
```

```
--- exec:3.1.0:exec (default-cli) @ DataStructure ---
Element 30 found at index 2
-----
BUILD SUCCESS
-----
Total time: 0.891 s
Finished at: 2024-08-18T10:01:12+05:00
-----
```

## Update Operation in an Array

The update operation in an array involves changing the value of an element at a specific index. Since arrays allow random access, updating an element is a straightforward process and is performed in constant time.

```
package com.mycompany.datastructure;
public class DataStructure {
    public static void main(String[] args) {
        // Step 1: Declare and initialize an array
        int[] arr = {10, 20, 30, 40, 50};

        // Step 2: Print the array before the update
        System.out.println("Array before update:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }

        // Step 3: Update the element at index 2
        int updateIndex = 2;
        int newValue = 35;

        if (updateIndex >= 0 && updateIndex < arr.length) {
            arr[updateIndex] = newValue;
        } else {
            System.out.println("Invalid index");
        }

        // Step 4: Print the array after the update
        System.out.println("\nArray after update:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }
    }
}
```

```
--- exec:3.1.0:exec (default-cli) @ DataStructure ---
```

```
Array before update:
```

```
Element at index 0: 10
```

```
Element at index 1: 20
```

```
Element at index 2: 30
```

```
Element at index 3: 40
```

```
Element at index 4: 50
```

```
Array after update:
```

```
Element at index 0: 10
```

```
Element at index 1: 20
```

```
Element at index 2: 35
```

```
Element at index 3: 40
```

```
Element at index 4: 50
```

```
-----  
BUILD SUCCESS
```

```
-----  
Total time: 1.286 s
```

```
Finished at: 2024-08-18T10:06:05+05:00  
-----
```