LINEAR DATA STRUCTURES

A **linear data structure** is a data structure where elements are organized **sequentially**, wherein each element is positioned in a linear order, and traversal typically follows one direction (from start to end).

Static and Dynamic Arrays

As mentioned, an **array** is a fixed-length, ordered collection of values of the same type stored in contiguous memory locations. The collection may be ordered in several dimensions.

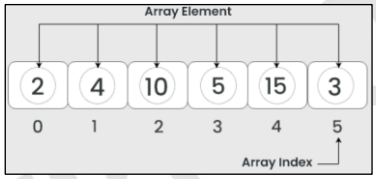
An array can be considered as the simplest type of data structure, and can be either a one-dimensional array or a two-dimensional array (matrix). They are used to implement tables, especially lookup tables, and lists and strings used by almost every program on a computer.

It can be compared to a series of containers or slots lined up in a row, with each slot containing **elements** (values or variables). The elements are identified by at least one (1) **array index** or **key**.

In programming, here's how an array looks:

```
int[] array = {2, 4, 10, 5, 15, 3};
```

Visually, this is how it looks:



In the example, `array[0]` holds the value of 2, `array[1]` holds the value of 4, and so on.

These are some basic Array operations in Java:

Java Operation	Syntax
Array Declaration	`type[] arrayName;` **type**: The data type of the array elements (e.g., int, String). **arrayName**: The name of the array.
Create an Array Using the **new** keyword	*// Creating an array of 4 integers* `int[] numbers = new int[4];` This statement initializes the numbers array to hold 5 integers. The default value for each element is 0.
Access an Element of an Array	*// Setting the first element of the array* `numbers[0] = 5;` *// Accessing the first element* `int firstElement = numbers[0];` The first line sets the value of the first element to 5. The second line retrieves the value of the first element.

Java Operation	Syntax
Change an Array Element	To change an element, assign a new value to a specific index: *// Changing the first element to 20* numbers[0] = 20;
Array Length Using the **length** property	*// Getting the length of the array* `int length = numbers.length;`

Static Array

Static arrays are data structures with a fixed size. Once an array is created, its size cannot be changed or altered. Here are its key features:

- **Fixed Size**: The size of the array is set during the declaration and cannot be modified.
- **Memory Allocation**: The memory is located on the stack, which is faster but has a limited size.
- **Access Time**: The elements can be accessed in constant time since the address of the elements can be calculated using the base address and the index.
- **Usage**: It is ideal for scenarios where the number of elements is known beforehand, such as storing fixed-sized data like days of the week and months of the year.

In Java, static arrays are created with a fixed size using the new keyword. The size of the array is specified when the array is initialized. A static array is declared as follows:

```
//Static array of size 6
int[] staticArrayName = new int [6];

//Assigning value to the first element
staticArrayName[0] = 15;
```

Here is a sample Java program:

```
public class StaticArrayDeclareAssign {
    public static void main(String[] args) {
        // Declare a static array of size 3
        int[] numbers = new int[3];

        // Assign values to each index
        numbers[0] = 5;
        numbers[1] = 10;
        numbers[2] = 15;

        // Display the values
        System.out.println("Array values:");
        System.out.println("Index 0: " + numbers[0]);
        System.out.println("Index 1: " + numbers[1]);
        System.out.println("Index 2: " + numbers[2]);
    }
}
```

Output:

```
Array values:

Index 0: 5
Index 1: 10
Index 2: 15
```

Dynamic Array

Dynamic arrays, or resizable arrays, can change size during runtime. Here are its key features:

- **Resizable**: The array can be resized, allowing it to grow or shrink based on the requirements
- **Memory Allocation**: The memory is allocated on the heap, which is more flexible but can be slower due to the overhead of dynamic memory management.
- **Access Time**: Similar to static arrays, elements can be accessed in constant time after they are created.
- **Usage**: It is useful when the number of elements is not known in advance or when the array needs to expand during program execution.

In Java, `ArrayList` is used to create dynamic arrays that can grow and shrink at runtime. A dynamic array is declared as follows:

```
//Dynamic array
import java.util.ArrayList;
ArrayList<Integer> dynamicArray = new ArrayList<>();

//Adding an element to the array
dynamicArray.add(10);
```

Here is a sample Java program:

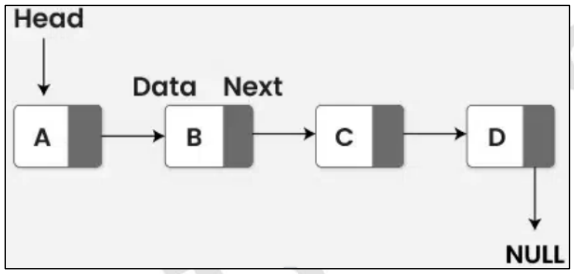
```
import java.util.ArrayList;

public class DynamicArrayExample {
    public static void main(String[] args) {
        // Declare a dynamic array of integers
        ArrayList<Integer> numbers = new ArrayList<>();

        // Add elements to the array
        numbers.add(5);
        numbers.add(10);
        numbers.add(15);

        // Display the elements
        System.out.println("Dynamic Array Elements:");
        for (int number : numbers) {
            System.out.println(number);
        }
    }
}
```

Output:

```
Dynamic Array Elements:
5
10
15
```

Fundamentals of Linked Lists

A **linked list** consists of **chains of nodes**, where each node contains information, such as **data**, and a **pointer to the next node**. Each node is composed of **data** and a **reference** (a link) to the next node in the sequence; more complex variants add additional links.



In Java, the syntax to define a Linked List is as follows:

```
LinkedList<data_type> linkedListName = new LinkedList<data_type>();
```

where `data_type` is the type of elements stored in the linked list, and `linkedListName` is the name of the linked list.

A sample Java program:

```
import java.util.LinkedList;

class Main {
  public static void main(String[] args){
    // Create linked list
    LinkedList<String> colors = new LinkedList<>();

    // Add elements to LinkedList
    colors.add("Yellow");
    colors.add("Orange");
    colors.add("Red");
    System.out.println("LinkedList: " + colors);
  }
}
```

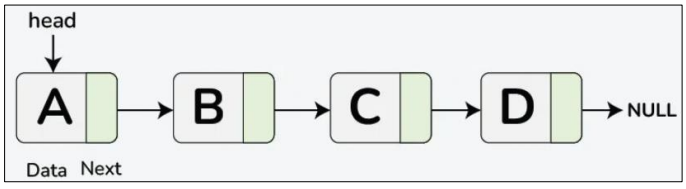
Output:

```
LinkedList: [Yellow, Orange, Red]
```

Singly Linked List

A **singly linked list** consists of **nodes** where each node contains a **data field** and a **reference** to the next node in the linked list. The next of the last node is **null**, indicating the end of the list.

In a singly linked list, each node has two parts: **data** and a **pointer** to the next node. This allows nodes to be dynamically linked together, forming a chain-like sequence.



The syntax of a singly linked list in Java is shown in this **Node Class** which represents a single element (node) in the list.

```
class Node {
    int data; // data stores the value
    Node next; // next refers to the next node in the list

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

When used, this is how the singly linked list works in Java:

```
public class TestNode {
    public static void main(String[] args) {
        Node n = new Node(100);
        System.out.println("Data: " + n.data);
        System.out.println("Next: " + n.next);
    }
}
```

Explanation:

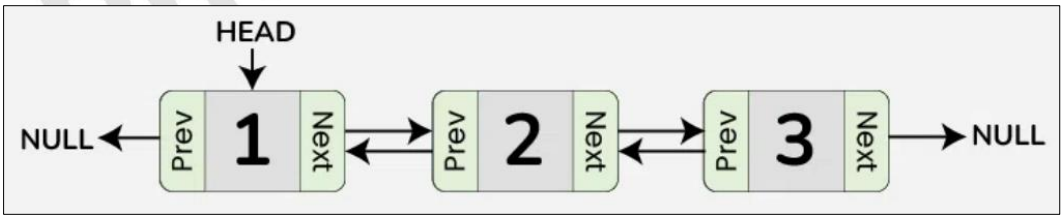
- Declares a Node object named n
  - `Node n = new Node(100);`
  - Creates a node with `data = 100` and `next = null`
- Prints the `data` of the node
  - `System.out.println("Data: " + n.data);`
  - Outputs: `Data: 100`
- Prints the `next` reference of the node
  - `System.out.println("Next: " + n.next);`
  - Outputs: `Next: null` (since it is not connected to any other node)

Output:

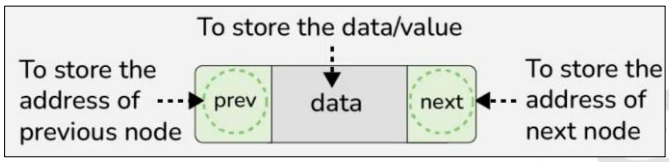
```
Data: 100
Next: null
```

Doubly Linked List

A **doubly linked list** is a more complex data structure than a singly linked list. It allows for efficient traversal of the list in both directions, as each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.



In data structures, a doubly linked list is represented by nodes with three (3) fields, which are: data, a pointer to the next node (*next*), and a pointer to the preview node (*prev*).



The syntax of a doubly linked list in Java is shown in the **Node** class.

```
class Node {
    int data; // data stores the value
    Node next; // next points to the next node
    Node prev; // points to the previous node

    Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
```

When used, this is how the doubly linked list works in Java:

```
// Main class to test the doubly linked list
public class TestDLL {
    public static void main(String[] args) {
        // Create a new doubly linked list
        DoublyLinkedList list = new DoublyLinkedList();

        // Insert three elements
        list.insert(1);
        list.insert(2);
        list.insert(3);

        // Display elements from head to tail
        list.displayForward();

        // Display elements from tail to head
        list.displayBackward();
    }
}

// Doubly linked list class definition
class DoublyLinkedList {

    // Node class for each element in the doubly linked list
    private static class Node {
        int data;       // Data held by the node
        Node prev, next; // Pointers to previous and next nodes

        // Constructor initializes node with given data
        Node(int data) {
```

```
            this.data = data;
        }
    }

    private Node head, tail; // References to the head and tail of the list

    // Method to insert a new node at the end of the list
    public void insert(int value) {
        Node n = new Node(value); // Create a new node with given value

        if (head == null) {
            // If list is empty, head and tail are both the new node
            head = tail = n;
        } else {
            // Link the new node to the end of the list
            tail.next = n;
            n.prev = tail;
            tail = n; // Update tail to point to the new last node
        }
    }

    // Method to display the list from head to tail
    public void displayForward() {
        Node cur = head; // Start from the head
        while (cur != null) {
            System.out.print(cur.data + " <-> "); // Print current node's data
            cur = cur.next; // Move to next node
        }
        System.out.println("null"); // End of list
    }

    // Method to display the list from tail to head
    public void displayBackward() {
        Node cur = tail; // Start from the tail
        while (cur != null) {
            System.out.print(cur.data + " <-> "); // Print current node's data
            cur = cur.prev; // Move to previous node
        }
        System.out.println("null"); // Start of list
    }
}
```

Output:

```
Forward: 1 <-> 2 <-> 3 <-> null
Backward: 3 <-> 2 <-> 1 <-> null
```

Explanation:

- `insert(1)`, `insert(2)`, and `insert(3)` build a doubly linked list like:

  `[1] <-> [2] <-> [3]`

- `displayForward()` starts from `head` that traverses with `.next`
- `displayBackward()` starts from the **last node** that traverses with `.prev`

Linked List Operations

Various linked list operations allow performing different actions on linked lists.

Here are some basic linked list operations:

- **Traversal** – accesses each element of the linked list.
- **Insertion** – adds a new element to the linked list.
- **Deletion** – removes the existing elements.
- **Search** – finds a node in the linked list.
- **Sort** – sorts the nodes of the linked list.

Traversal

It means visiting each node in the linked list one by one to read, display, or process its data. In Java, traversal is typically done using a **while loop** that follows the next references from the **head** to the **end** of the list.

The program below demonstrates how to **traverse a singly linked list** in Java:

```
class Node {
    int data;
    Node next;

    Node(int d) {
        data = d;
        next = null;
    }
}

public class LinkedListTraversal {
    public static void main(String[] args) {
        // Create a linked list: 10 -> 20 -> 30
        Node head = new Node(10);
        head.next = new Node(20);
        head.next.next = new Node(30);

        // Traverse and display
        Node current = head;
        System.out.println("Traversing the linked list:");
        while (current != null) {
            System.out.println(current.data);
            current = current.next;
        }
    }
}
```

Output:

```
Traversing the linked list:
10
20
30
```

Insertion

It means adding a new node to a linked list. It's one of the core operations supported by all types of linked lists.

The program below demonstrates how insertion works in Java:

```
class Node {
    int data;
    Node next;
    Node(int value) {
        data = value;
        next = null;
    }
}
public class LinkedListDemo {
    public static void main(String[] args) {
        Node head = new Node(10);
        head.next = new Node(20);

        Node newNode = new Node(30);
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;

        // Display the list
        temp = head;
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");
    }
}
```

Output:

```
10 -> 20 -> 30 -> null
```

Deletion

It is the operation of removing a node from a linked list. In Java (singly linked list), it involves:

- Finding the node to delete
- Adjusting `.next` pointers
- Ensuring memory references are safely discarded

Search

It means finding a node that contains a specific value by traversing each node **one-by-one** starting from the head.

- Start from the `head` node
- Compare the target value with `current.data`
- If matched → return found
- If not → move to `current.next`

- Repeat until either:
  - Value is found, or
  - End of list (current == null)

Sort

It means arranging its nodes' values in ascending or descending order. Unlike arrays, linked lists do not support random access, so sorting must be done by rearranging the node links or values manually, not by indexing.

The program below demonstrates sorting Java's built-in LinkedList.

```
import java.util.Collections;
import java.util.LinkedList;

public class BuiltInSort {
    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        list.add(40);
        list.add(10);
        list.add(30);

        System.out.println("Before: " + list);
        Collections.sort(list);
        System.out.println("After: " + list);
    }
}
```

Output:

```
Before: [40, 10, 30]
After: [10, 30, 40]
```


References:

Galgotias University. (2020). *Arrays.* Retrieved on June 26, 2025, from https://www.gcjana.in/courses/galgotiasuniversity/oddsem2024/ppts/Lce-2-Arrays.pdf

GeeksforGeeks. (2025). *Introduction to linear data structures.* Retrieved on June 26, 2025, from https://www.geeksforgeeks.org/java/arrays-in-java/

Oracle. (n.d.a.). Chapter 10. Arrays. Retrieved on June 26, 2025, from https://docs.oracle.com/javase/specs/jls/se7/html/jls-10.html