

ESTRUCTURA DE DATOS:

¿Qué es una estructura de datos?

Una **estructura de datos** es una forma de organizar, gestionar y almacenar datos para que puedan ser utilizados de manera eficiente. Dependiendo del tipo de problema, se escoge la estructura más adecuada para optimizar el tiempo y uso de memoria.

♦ Tipos primitivos vs. estructuras de datos

- **Tipos primitivos:** Son los tipos básicos que ofrece el lenguaje, como `int`, `float`, `char`, `boolean`, etc.
- **Estructuras de datos:** Son contenedores que permiten almacenar múltiples valores y organizarlos. Ej: `ArrayList`, `Stack`, `Queue`, `HashSet`, `HashMap`.

☒ Estructuras de Datos Comunes en Java

1. **Listas (`ArrayList`)**
 - a. Permite almacenar elementos en orden, acceder por índice y duplicados.
2. **Pilas (`Stack`)**
 - a. Estructura LIFO (último en entrar, primero en salir).
3. **Colas (`Queue`)**
 - a. Estructura FIFO (primero en entrar, primero en salir).
4. **Conjuntos (`HashSet`)**
 - a. Almacena elementos únicos, sin orden específico.
5. **Diccionarios (`HashMap`)**
 - a. Almacena pares clave-valor, claves únicas.

Clasificación de las estructuras de datos

1. Lineales

Los elementos están en una secuencia y cada elemento tiene un sucesor y/o antecesor.

Estructura	Características
Array / Arreglo	Tamaño fijo, acceso por índice, eficiente para lectura.

Lista enlazada	Cada elemento apunta al siguiente. Tamaño dinámico.
Pila (Stack)	LIFO: el último en entrar es el primero en salir.
Cola (Queue)	FIFO: el primero en entrar es el primero en salir.
Deque (Double-ended queue)	Inserción/eliminación en ambos extremos.

2. No lineales

Los elementos no están en una secuencia lineal. Se usan para representar relaciones jerárquicas o conexiones complejas.

Estructura	Características
Árboles (Trees)	Representan relaciones jerárquicas. Ej: árbol binario, árbol AVL.
Grafos (Graphs)	Conjunto de nodos conectados por aristas. Pueden ser dirigidos/no dirigidos.

3. Estructuras asociativas (clave-valor)

Estructura	Características
HashMap	Asociación entre claves únicas y valores. Acceso rápido si no hay colisiones.
HashSet	Similar al HashMap, pero solo guarda claves (sin valores).

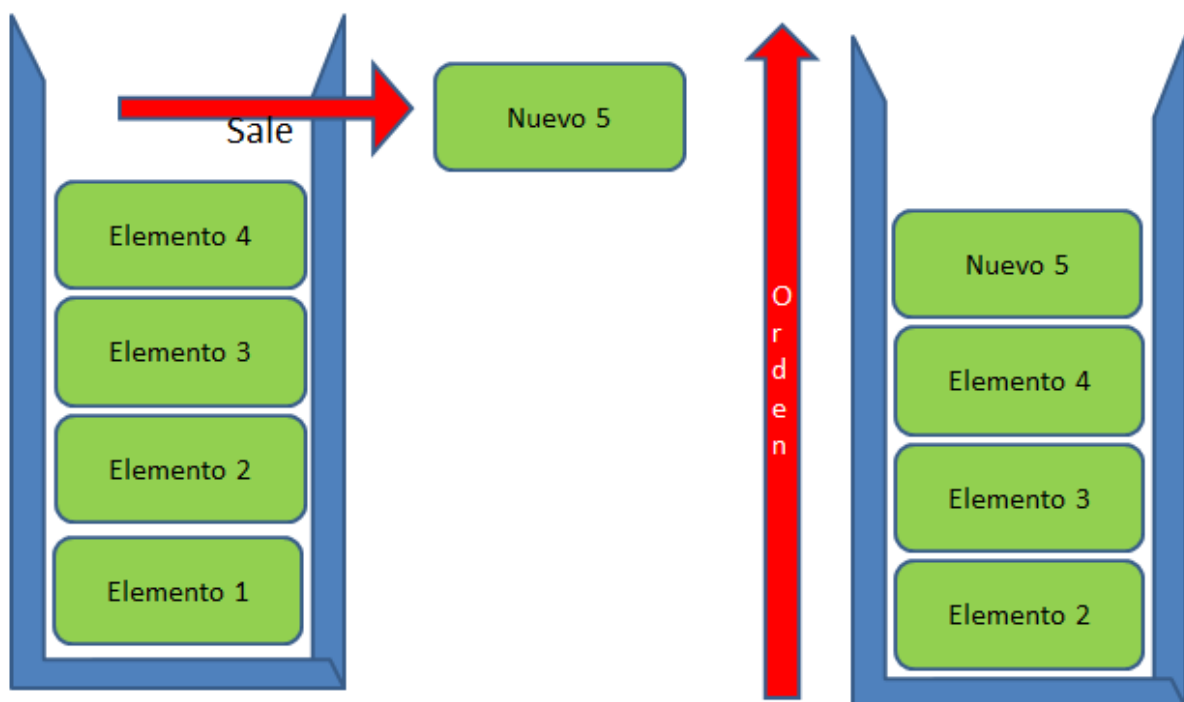
Comparación rápida

Estructura	Inserción	Búsqueda	Eliminación	Uso típico
ArrayList	Rápida (al final)	Rápida (índice)	Lenta (intermedia)	Listas ordenadas
LinkedList	Rápida	Lenta (requiere recorrer)	Rápida	Manipulación frecuente de elementos
Stack	Rápida	LIFO	Rápida	Deshacer acciones
Queue	Rápida	FIFO	Rápida	Procesos en orden
HashMap	Muy rápida	Muy rápida	Muy rápida	Almacenar configuración, cachés
HashSet	Muy rápida	Muy rápida	Muy rápida	Eliminar duplicados

🧠 ¿Cuándo usar cada una?

- **Usá una pila (Stack)** cuando necesites deshacer acciones (como en editores).
- **Usá una cola (Queue)** cuando se atiende en orden de llegada (sistemas de tickets).
- **Usá un ArrayList** si querés acceso por índice y pocos cambios internos.
- **Usá una LinkedList** si vas a insertar/eliminar mucho en el medio de la lista.
- **Usá un HashMap** si necesitás acceder a datos por clave (como un diccionario).
- **Usá un HashSet** si querés evitar duplicados rápidamente.

Pila (Stack) – LIFO LAST IN FIRST OUT



Cola (Queue) – FIFO: FIRST IN FIRST OUT

Lista Enlazada (Singly Linked List)

`hashCode()` y `equals()` son dos métodos clave en Java que determinan cómo se comparan y almacenan objetos en estructuras como `HashMap`, `HashSet` y `Hashtable`.

◇ `equals(Object obj)`

Este método **compara si dos objetos son “lógicamente iguales”**.

Por defecto (en la clase `Object`), compara por **referencia** (es decir, si apuntan al mismo objeto en memoria), pero podés sobrescribirlo para comparar el contenido.

Ejemplo:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
    Persona persona = (Persona) obj;
    return nombre.equals(persona.nombre) && edad == persona.edad;
}
```

hashCode()

Devuelve un número entero que representa el “**código hash**” del objeto. Este número se usa para **determinar en qué posición se almacenará el objeto** en una tabla hash (HashMap, HashSet).

Reglas importantes:

1. Si dos objetos son iguales según `equals()`, deben tener el mismo `hashCode()`.
2. No es obligatorio que dos objetos con el mismo `hashCode()` sean iguales, pero ayuda si lo son.

Ejemplo:

```
@Override
public int hashCode() {
    return Objects.hash(nombre, edad); // Usá campos usados en equals()
}
```

¿Por qué son importantes juntos?

En estructuras como HashSet:

- Primero usa `hashCode()` para buscar en una "cubeta" (bucket).
- Luego usa `equals()` para ver si realmente es igual.

Si no sobrescribís ambos correctamente, podés tener errores como:

- Objetos duplicados que no se eliminan.
- Objetos que no se encuentran aunque estén ahí.

Ejemplo Completo:

```
import java.util.Objects;

public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Persona)) return false;
        Persona persona = (Persona) o;
        return edad == persona.edad && nombre.equals(persona.nombre);
    }

    @Override
    public int hashCode() {
        return Objects.hash(nombre, edad);
    }
}
```

🔑 1. Costos de operaciones (complejidad temporal)

🔍 ¿Qué es la "complejidad temporal"?

Es una forma de **medir cuánto tarda una operación** en función de la cantidad de datos que tenés.

Se expresa con **notación Big O**


Notación	Significado
$O(1)$	Tiempo constante. Siempre tarda lo mismo. 🔥 Muy eficiente.
$O(n)$	Tiempo lineal. Tarda más cuanto más grande es la estructura.

$O(\log n)$	Tiempo logarítmico. Crece muy lento (ej: búsqueda binaria).
$O(n^2)$	Tiempo cuadrático. Mucho más lento (ej: algoritmos burbuja).

¿Cómo afecta esto a las estructuras de datos?

Cada estructura permite hacer ciertas operaciones **más rápido que otras**.
Las más comunes son:

- **Acceso:** Obtener un valor por su posición o clave.
- **Búsqueda:** Encontrar si un valor está.
- **Inserción:** Agregar un nuevo elemento.
- **Eliminación:** Borrar un elemento.

 Tabla con ejemplos explicados

Estructura	Acceso	Búsqueda	Inserción	Eliminación	¿Por qué?
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	Accedés por índice, pero insertar o eliminar requiere mover cosas.
ArrayList	$O(1)$	$O(n)$	$O(n)$	$O(n)$	Igual que el array, pero dinámico.
LinkedList	$O(n)$	$O(n)$	$O(1)^*$	$O(1)^*$	Agregar/quitar al inicio es rápido, pero acceder tarda.
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	Solo se trabaja con el tope.
Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$	Solo al inicio/final.
HashMap	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Usa hashing para ubicar claves rápidamente.
HashSet	$O(1)$	$O(1)$	$O(1)$	$O(1)$	Similar a HashMap pero sin valores.

Ejemplo práctico

Imaginá una lista con 1.000.000 de elementos:

- Si usás ArrayList y querés insertar un nuevo elemento al principio (index 0), **Java debe mover todos los demás un lugar** $\rightarrow O(n)$.

- En cambio, si usás `LinkedList`, puede **agregar al principio sin mover nada** $\rightarrow O(1)$.

🎯 ¿Para qué sirve esto?

Saber estos costos te ayuda a:

- Elegir la estructura correcta.
- Hacer programas más rápidos y eficientes.
- Superar entrevistas técnicas que te preguntan:
"¿Cuál es la complejidad de esta solución?"

🔑 2. Operaciones que cada estructura facilita

- **Listas:** recorrer secuencias, modificar elementos.
- **Pilas:** deshacer operaciones, retroceso en navegación.
- **Colas:** sistemas de turnos, buffers.
- **HashMaps:** buscar por clave, contar frecuencia.
- **Conjuntos (HashSet):** eliminar duplicados, ver si un elemento está.
- **Árboles:** búsqueda ordenada, jerarquías, autocompletado.
- **Grafos:** rutas, redes, conexiones.

📁 3. Árboles y variantes importantes

Te conviene familiarizarte con:

- **Árbol binario:** cada nodo tiene hasta 2 hijos.
- **Árbol binario de búsqueda (BST):** hijos izquierdos $<$ nodo $<$ hijos derechos.
- **AVL o Red-Black Tree:** árboles balanceados (mantienen eficiencia).
- **Heap (montículo):** árbol completo, útil para obtener el mínimo o máximo.

4. Recorridos típicos

En árboles:

- **Inorden (in-order):** izquierda → raíz → derecha → (útil en BST).
- **Preorden (pre-order):** raíz → izquierda → derecha.
- **Postorden (post-order):** izquierda → derecha → raíz.
- **Nivel por nivel (BFS):** con cola.

5. Estructuras avanzadas que podrían aparecer:

- **Trie (árbol de prefijos):** ideal para autocompletado o buscar palabras.
- **Grafos:** con listas de adyacencia o matrices de adyacencia.
- **Tabla Hash con colisiones (hash chaining, open addressing).**
- **Deque (double-ended queue):** permite operaciones en ambos extremos.

6. Buena práctica: elegir la estructura correcta

Siempre pensá:

1. ¿Necesito acceder rápido por índice? → ArrayList.
2. ¿Necesito insertar/eliminar frecuentemente? → LinkedList.
3. ¿Debo mantener elementos únicos? → HashSet.
4. ¿Clave-valor? → HashMap.
5. ¿Orden natural? → TreeSet, TreeMap.

7. Preparación para entrevistas / exámenes

Aprendete patrones comunes:

- Invertir una lista.

- Validar paréntesis con pila.
- Detectar ciclos en listas o grafos.
- Usar cola para BFS.
- HashMap para contar ocurrencias.
- Sliding window y two pointers.

☑ Temas esenciales de estructuras de datos en Java

◇ 1. Colecciones de Java (`java.util`)

Java tiene una biblioteca muy completa con estructuras listas para usar:

Interfaz / Clase	¿Qué representa?
List	Lista ordenada, permite duplicados (ArrayList, LinkedList)
Set	Colección sin duplicados (HashSet, TreeSet, LinkedHashSet)
Map	Pares clave-valor (HashMap, TreeMap, LinkedHashMap)
Queue	Cola FIFO (LinkedList, ArrayDeque, PriorityQueue)
Deque	Cola doble extremo (ArrayDeque)

◇ 2. Clases concretas más importantes

- ArrayList vs. LinkedList: diferencias de rendimiento.
- HashMap vs. TreeMap: sin orden vs. ordenado por clave.
- HashSet vs. TreeSet: sin orden vs. orden natural.
- PriorityQueue: estructura de **heap** para obtener el mínimo o máximo.

◇ 3. Interfaces funcionales y Streams (útil para listas/mapas)

Aunque no es estructura en sí, es muy útil para trabajar con colecciones.

```
list.stream()
    .filter(x -> x % 2 == 0)
    .sorted()
    .forEach(System.out::println);
```

Esto **simplifica mucho** recorrer, filtrar o transformar estructuras.

◇ 4. Iteradores

- for-each: ideal para recorrer.
- Iterator: si necesitas eliminar mientras recorres.

◇ 6. Comparators y Comparable

Para ordenar listas o usar estructuras ordenadas como TreeSet, PriorityQueue.

```
class Persona implements Comparable<Persona> {
    public int compareTo(Persona otra) {
        return this.edad - otra.edad;
    }
}
```

O bien con `Comparator` :

```
java

list.sort(Comparator.comparing(Persona::getEdad));
```

◇ 8. Excepciones comunes

- NullPointerException al acceder a elementos nulos.
- IndexOutOfBoundsException en ArrayList.
- ConcurrentModificationException al modificar mientras iteras.

