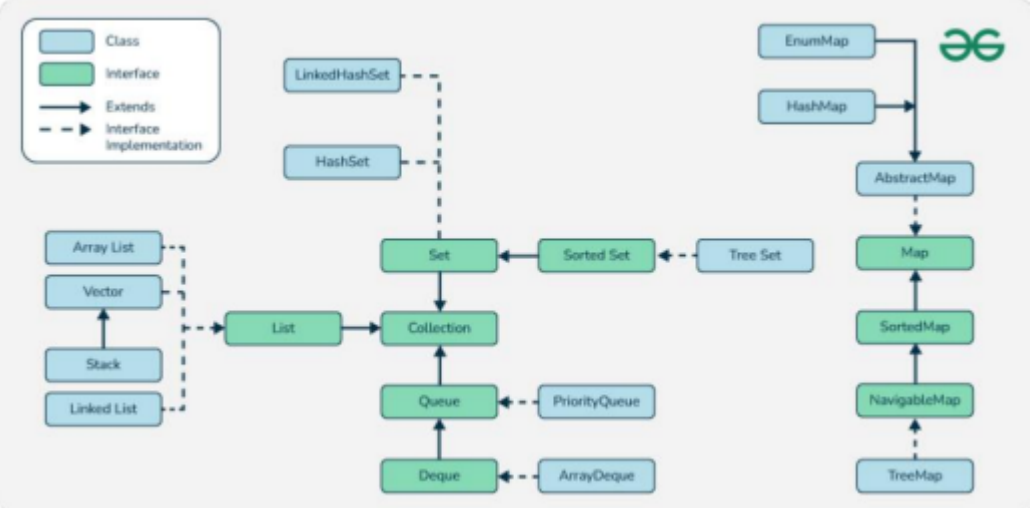
1. Introducción

Las colecciones permiten gestionar grupos de datos: almacenarlos, recuperarlos y manipularlos. Son un conjunto de interfaces, clases concretas y abstractas que definen tipos de datos abstractos en Java que implementan colecciones de elementos. Por ejemplo: *listas, pilas, colas, conjuntos* y *mapas*.

Figura 1: Mapa mental de las diferentes estructuras



Imagínese que es un bibliotecario responsable de gestionar una gran cantidad de libros. Necesita un sistema para realizar un seguimiento eficiente de todos estos libros. En Java, el concepto de colecciones cumple una función similar. Las colecciones son como el equivalente digital de su sistema de organización en la biblioteca. Le permiten agrupar y administrar objetos de una manera estructura y flexible.

Tarea: observando la figura, ¿existen instancias de objetos que no se crearían? Da algunos ejemplos.

Como puedes imaginar, las colecciones pueden tener un orden determinado. Este orden puede ser lineal o no lineal, por lo que existen diferentes estructuradas para que podamos utilizar cada una de ellas cuando más lo necesitamos.

Además, cuando se trabaja con aplicaciones Java del mundo real, a menudo se trabaja con grandes cantidades de datos, que es necesario organizar y acceder a ellos de forma eficiente. Sin colecciones, tendría que escribir una gran cantidad de código personalizado para administrar estructuras de datos, lo cual sería difícil de mantener y de escalar.

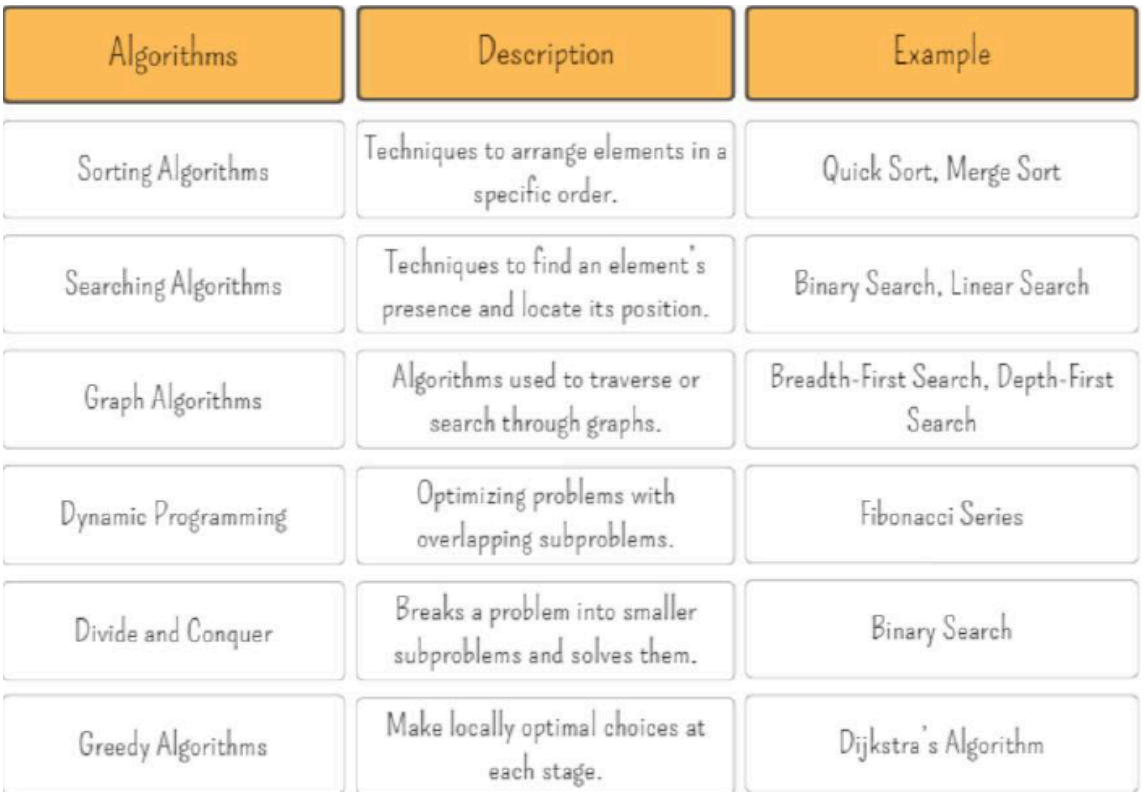
Las colecciones de Java proporcionan una forma estandarizada de gestionar estos datos. Cuentan con métodos integrados que simplifican tareas complejas como la ordenación, la búsqueda y el filtrado. De hecho, las colecciones son dinámicas, por lo que pueden crecer o reducirse según sea necesario, lo que supone una ventaja significativa con respecto a las matrices que tienen un tamaño fijo.

Figura 2: Estructuras de datos

Data Structures	Description
Array	Ordered collection of elements of the same data type.
Linked List	Dynamic data structure where elements are connected through nodes.
Stack	Follows Last In, First Out (LIFO) principle.
Queue	Follows First In, First Out (FIFO) principle.
Tree	Hierarchical data structures with a root node and branches.
Graph	Consists of nodes (vertices) and edges representing connections.

Tarea: busca en Internet la representación gráfica de las estructuras de datos que hemos visto, para que puedas hacerte una idea de cómo se maneja la información y su estructura.

Figura 3: Algoritmos



Algorithms	Description	Example
Sorting Algorithms	Techniques to arrange elements in a specific order.	Quick Sort, Merge Sort
Searching Algorithms	Techniques to find an element's presence and locate its position.	Binary Search, Linear Search
Graph Algorithms	Algorithms used to traverse or search through graphs.	Breadth-First Search, Depth-First Search
Dynamic Programming	Optimizing problems with overlapping subproblems.	Fibonacci Series
Divide and Conquer	Breaks a problem into smaller subproblems and solves them.	Binary Search
Greedy Algorithms	Make locally optimal choices at each stage.	Dijkstra's Algorithm

Tarea: busca en Internet una representación en vídeo de cómo funcionan los algoritmos que hemos visto. Observa las diferencias en cada uno de ellos y date cuenta de que todos tienen una finalidad y deben utilizarse para un caso concreto.

1. Estructuras de datos

Las estructuras de datos se utilizan para organizar y almacenar datos en una computadora. Nos ayudan a mantener la información en orden y facilitan su uso. Piense en las estructuras de datos como si fueran diferentes tipos de cajas o contenedores para sus cosas.

¿Qué son las estructuras de datos?

- Las estructuras de datos son formatos especiales para organizar y almacenar datos
- Son como contenedores que contienen información de una manera específica.
- Los ejemplos incluyen listas, pilas y árboles

¿Por qué son importantes?

- Ayuda a administrar y organizar grandes cantidades de datos
- Facilitar y agilizar la búsqueda y el uso de la información
- Permitir que las computadores trabajen de manera más eficiente
- Esencial para escribir programas informáticos buenos y rápidos

Comenzaremos con las más simples y pasaremos a estructuras más complejas. Para cada estructura de datos, también veremos sus implementaciones y complejidad temporal.

2.1 Arrays

Los arrays son una de las estructuras de datos más simples y comunes en Java. Son como una fila de cajas donde puedes almacenar cosas. Un array es una colección de elementos del mismo tipo, almacena elementos en orden, uno tras otro, cada elemento tiene un número (llamado índice) para encontrarlo fácilmente y tienen un tamaño fijo que no cambia después de su creación.

¿Qué hace el siguiente código?

```
int [] numbers = new int[5];

numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;

System.out.println("El tercer número es: " + numbers[2]);

for(int i = 0; i < numbers.length; i++){
   System.out.println("Número " + (i + 1) + " es: " + numbers[i]);
}
```

Los arrays son útiles cuando sabes cuántos elementos necesitas almacenar y cuándo se quiere acceder a los elementos rápidamente por su posición.

Complejidad de tiempo para operaciones de arrays:

- **Acceso:** O(1)
- **Búsqueda:** O(n)
- **Inserción:** O(n)
- **Eliminación:** O(n)

2.2 Listas

Las listas son como arrays mejoradas. Pueden crecer o reducirse según sea necesario. Java tiene dos tipos principales de listas: *ArrayList* y *LinkedList*.

2.2.1 ArrayList

Un ArrayList en Java es una estructura de datos redimensionable similar a una matriz que permite almacenar una lista de elementos. A diferencia de los arrays normales, el ArrayList puede almacenar ajustando su tamaño automáticamente cuando agregas o eliminar elementos, haciéndolo más flexible. Proporciona

métodos para agregar, eliminar y acceder fácilmente a elementos y mantiene el orden en el que se insertan los elementos.

Características de un ArrayList:

- Similar a una array, pero puede cambiar su tamaño
- Bueno para almacenar y acceder a elementos rápidamente
- No es bueno para agregar o quitar elementos en el medio

¿Qué hace el siguiente código?

```
import java.util.ArrayList;

public class ArrayListExample{
   public static void main (String[] args){
      ArrayList <String> fruits = new ArrayList <String>();
      fruits.add("Manzana");
      fruits.add("Plátano");
      fruits.add("Cereza");

      System.out.println("Frutas: " + fruits);
      System.out.println("Segunda fruta: " + fruits.get(1));
      fruits.set(0, "Albaricoque");
      fruits.remove("Cereza");
      System.out.println("Frutas actualizadas: " + fruits);
   }
}
```

Complejidad de tiempo para operaciones de ArrayList:

- **Acceso:** O(1)
- **Búsqueda:** O(n)
- **Inserción (al final):** O(1)
- **Inserción (índice específico):** O(n)
- **Eliminación (al final):** O(1)
- **Eliminación (índice específico):** O(n)

2.2.2 LinkedList

Una lista enlazada en Java es una estructura de datos que consta de una secuencia de elementos, donde cada elemento está vinculado al siguiente y al anterior, formando una cadena. A diferencia de una ArrayList, que almacena elementos en un bloque contiguo de memoria. Una LinkedList almacena cada elemento en un nodo separado que contiene el elemento en sí y las referencias a los nodos anterior y siguiente. Esto hace que la LinkedList sea eficiente para inserciones y eliminaciones en cualquier posición, pero más lento para acceder a elementos por índice.

Características de una LinkedList:

- Compuesto por elementos separados unidos entre sí
- Es útil para agregar o eliminar elementos en cualquier parte de la lista
- No es tan bueno para acceder a elementos por posición

¿Qué hace el siguiente código?

```
import java.util.LinkedList;

public class LinkedListExample{
   public static void main(String [] args){
      LinkedList <Integer> numbers = new LinkedList <Integer>();
      numbers.add(10);
      numbers.add(20);
      numbers.add(30);

      System.out.println("Números: " + numbers);
      numbers.addFirst(5);
      numbers.addLast(40);
      System.out.println("Números actualizados: " + numbers);
      numbers.removeFirst();
      System.out.println("Números definitivos: " + numbers);
```

Complejidad de tiempo para operaciones de LinkedList:

- **Acceso:** O(1)
- **Búsqueda:** O(n)
- **Inserción (principio/final):** O(1)
- **Inserción (índice específico):** O(n)
- **Eliminación (principio/final):** O(1)
- **Eliminación (índice específico):** O(n)

2.3 Pilas

Una pila en Java es una estructura de datos que sigue el principio de **último en entrar, primero en salir (LIFO)**. En Java, una pila permite operaciones como *Push* (agregar elementos), *Pop* (eliminar el elemento superior) y *Peek* (para comprobar el elemento superior sin eliminarlo). Se utiliza comúnmente para escenarios de retroceso, mecanismos de deshacer y análisis de expresiones. Por lo tanto, son útiles para tareas que necesitan revertir el orden o realizar un seguimiento de lo que sucedió por última vez.

Tarea: sin mirar el código, crear una clase Stack <T> que simule una pila utilizando un ArrayList <T>

Podemos crear una pila simple usando una lista de arreglos. A continuación se muestra un ejemplo:

```
import java.util.ArrayList;

public class Stack <T>{
   private ArrayList <T> stack;

   public Stack(){
      stack = new ArrayList<T>();
   }

   //Añade elementos a la pila
   public void push(T item){
       stack.add(item);
   }

   //Elimina y devuelve la cima de la pila
   public T pop(){
      if (isEmpty()){
         throw new IllegalStateException("Pila vacía");
      }
      return stack.remove(stack.size() - 1);
   }

   //Devuelve la cima de la pila
   public T peek(){
      if(isEmpty()){
         throw new IllegalStateException("Pila vacía");
      }
      return stack.get(stack.size() - 1);
   }

   //Comprueba si la pila está vacía
   public boolean isEmpty(){
      return stack.isEmpty();
   }

   //Devuelve el tamaño de la pila
   public int size(){
      return stack.size();
   }
```

Complejidad temporal de las operaciones con pilas:

- **Push:** O(1)
- **Pop:** O(1)
- **Peek:** O(1)
- **Búsqueda:** O(n)

2.4 Colas

Una cola es otra estructura de datos importante. Funciona como una fila de personas esperando algo. La primera persona que se une a la fila es la primera en salir. Sigue el principio **FIFO(First In First Out)**, es decir, el primer elemento que se

agrega, es el primero que se eliminará. En Java, una cola proporciona operaciones como agregar elementos, eliminar y devolver el primer elemento y ver el primer elemento sin eliminarlo. También es útil para administrar tareas en orden, como trabajos de impresión o solicitudes de servicio al cliente.

Tarea: sin mirar el código, crear una clase Queue<T> que simule una cola que utiliza una LinkedList <T>

Podemos crear una cola sencilla utilizando una LinkedList. A continuación, se muestra un ejemplo:

```
import java.util.LinkedList;

public class Queue <T>{
   private LinkedList<T> queue;

   public Queue(){
      queue = new LinkedList <T>();
   }

   //Añade elementos al final de la cola
   public void enqueue(T item){
      queue.addLast(item);
   }

   //Elimina y devuelve el elemento del principio de la cola
   public T dequeue(){
      if(isEmpty()){
         throw new IllegalStateException("Cola vacía");
       }
       return queue.removeFirst();
   }

   //Devuelve el elemento del principio de la cola
   public T peek(){
       if(isEmpty()){
          throw new IllegalStateException("Cola vacía");
       }
       return queue.getFirst();
   }

   //Comprueba si la cola está vacía
   public boolean isEmpty(){
      return queue.isEmpty();
   }

   //Devuelve el tamaño de la cola
   public int size(){
      return queue.size();
   }
```

Complejidad temporal de las operaciones con colas:

- **Enqueue:** O(1)
- **Dequeue:** O(1)
- **Peek:** O(1)
- **Búsqueda:** O(n)

2.5 Árboles

Un árbol en Java es una estructura de datos jerárquica en la que cada elemento, llamado **nodo**, está conectado a otros en una relación padre-hijo. El nodo superior se llama **raíz** y cada nodo puede tener 0 o más nodos secundarios. Los árboles son útiles para representar datos jerárquicos, como sistemas de archivos, estructuras organizativas o procesos de toma de decisiones. En Java existen tipos específicos de árboles, como los *árboles binarios*, en los que cada nodo tiene como máximo 2 hijos, y *árboles equilibrados* como **TreeMap** o **TreeSet**, que mantienen los elementos en orden ordenado para realizar búsquedas e inserciones rápidas.
Los árboles se utilizan comúnmente en búsquedas, ordenamientos y representaciones jerárquicas.

2.6 Árboles binarios de búsqueda (BST)

Es un tipo de árbol binario en el que cada nodo tiene como máximo 2 hijos y el árbol mantiene un orden específico. Para cualquier nodo dado, el hijo izquierdo contiene valores menores que el valor del nodo y del hijo derecho contiene valores mayores que el valor del nodo. Esta propiedad hace que las operaciones de búsqueda, inserción y eliminación sean eficientes, ya que permite delimitar rápidamente dónde se debe colocar o encontrar un valor. Los árboles binarios de búsqueda se utilizan comúnmente para buscar, ordenar y mantener un conjunto dinámico de datos ordenados.
Tarea: sigue las indicaciones del profesor para desarrollar tu primera implementación del árbol BST. Es muy importante entender cada uno de los pasos, luego habrá una actividad de implementación del BST
Complejidad de un **árbol binario de búsqueda** en Java:

- **Inserción:** O[log n] (En el mejor de los casos), O[log n] (En el caso promedio) y O[n] (En el peor de los casos).
- **Eliminación:** O[log n] (En el mejor de los casos), O[log n] (En el caso promedio) y O[n] (En el peor de los casos)
- **Búsqueda:** O[log n] (En el mejor de los casos), O[log n](En el caso promedio) y O[n] (En el peor de los casos)

2.7 Tablas hash

Una **tabla hash** en Java es una estructura de datos que almacena **pares clave-valor**, lo que permite recuperación rápida de valores en función de sus claves. Se utilizan ampliamente en muchas aplicaciones donde el acceso rápido a los datos es crucial. Si asignas varias claves al mismo índice se genera una colisión, para solucionar se emplean técnicas como **encadenamiento** o **direccionamiento abierto**.

Las tablas hash se implementan comúnmente en Java a través de clases como *HashMap* o *Hashtable*.

Esto es un ejemplo:

```
import java.util.HashMap;

public class HashMapExample{
   public static void main(String [] args){
      //Crear un nuevo HashMap
      HashMap <String,Integer> ages = new HashMap <>();

      //Añadir pares clave-valor al HashMap
      ages.put("Alice", 25);
      ages.put("Bob", 30);
      ages.put("Carol", 35);

      //Toma un valor del HashMap
      int bobAge = ages.get("Bob");
      System.out.println("Bob's age: " + bobAge);

      //Comprobar si una clave existe
      if(ages.containsKey("David")){
         System.out.println("David's age is known");
      }else{
         System.out.println("David's age is unknown");
      }

      //Actualiza un valor
      ages.put("Alice", 26);

      //Elimina un par de clave-valor
      ages.remove("Carol");
```

HashMap es muy eficiente para almacenar y recuperar datos cuando se tiene una clave única para cada dato. Es particularmente útil cuando se necesita buscar valores con frecuencia en función de sus claves.

Complejidad temporal de **HashMap** en Java:

- **Insertar:** O(1) caso promedio, O(1) peor caso
- **Eliminar**: O(1) caso promedio, O(1) peor caso
- **Búsqueda:** O(1) caso promedio, O(1) peor caso

Tarea: vamos a hacer una pequeña guía telefónica utilizando tablas hash. Escuche los detalles de la implementación que ofrece el docente para tener una idea de qué hacer con cada método.