



# Argentina Programa





# Clase 20: Estructura de datos - Generics - Arrays - List



# Agenda



Familiarizarse con los conceptos básicos  
relacionados a Estructura de datos - Generics -  
Arrays - List

- Estructura de datos - Generics - Arrays - List
- Práctica Estructura de datos - Generics - Arrays - List.



# Estructura de datos



Las estructuras de datos en Java son conjuntos de valores, variables y tipos de datos organizados para soportar operaciones específicas como agregar y eliminar elementos, buscar elementos, ordenar elementos, etc.





## Clasificación de estructuras de datos:

Las **estructuras de datos estáticas** son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del programa, mientras que una **estructura de datos dinámica** es aquella en la que el tamaño ocupado en memoria puede modificarse durante la ejecución del programa.

Las **estructuras de datos lineales** son aquellas en las que los elementos de datos se organizan en una secuencia lineal, es decir, los datos se almacenan en una serie continua de posiciones de memoria.

Las **estructuras de datos no lineales** no tienen una estructura secuencial lineal. Estas estructuras de datos se organizan de manera jerárquica y pueden tener varios elementos de datos relacionados.



# Estructura de datos



Algunas de las estructuras de datos más comunes en Java incluyen:

- Arrays
- Listas
- Pilas
- Colas
- Conjuntos -Tablas hash-
- Mapas





En Java, los tipos genéricos permiten escribir código que puede ser utilizado con diferentes tipos de datos, sin necesidad de especificarlos en el momento de escribirlo. Esto mejora la reutilización del código y reduce los errores de tiempo de compilación.

Los tipos genéricos se declaran con angle brackets `<>` y pueden ser utilizados en clases, interfaces y métodos.





Fueron introducidos en Java 5 y permiten a los desarrolladores crear clases, interfaces y métodos que pueden trabajar con cualquier tipo de datos, sin tener que especificar exactamente qué tipo de datos se utilizará.

Esto significa que pueden ser utilizados para crear colecciones de objetos, estructuras de datos y algoritmos, entre otros, que sean genéricos y sean útiles para muchos tipos diferentes de datos.

# Generics



Las ventajas del uso de generics en Java incluyen:

- Seguridad de tipos: los generics permiten detectar errores de tipo en tiempo de compilación, lo que ayuda a reducir los errores de programación.
- Reutilización de código: los generics permiten crear clases y métodos genéricos que pueden ser utilizados con diferentes tipos de datos, lo que reduce la necesidad de escribir código duplicado.
- Eficiencia: los generics permiten evitar la conversión de tipos en tiempo de ejecución, lo que puede mejorar el rendimiento del programa.

Las desventajas del uso de generics en Java incluyen:

- Curva de aprendizaje: los generics pueden ser complicados de entender al principio, especialmente si no se tiene experiencia en programación orientada a objetos.
- Mayor complejidad: el uso de generics puede hacer que el código sea más complejo y difícil de leer en algunos casos.
- Problemas de compatibilidad: en algunos casos, el uso de generics puede causar problemas de compatibilidad con versiones anteriores de Java o con otras bibliotecas que no utilizan generics.





## Ejemplo

**Contenedores de datos:** Por ejemplo, se puede crear una clase genérica MyList que almacene una lista de objetos de cualquier tipo:

```
public class MyList<T> {  
    private List<T> list = new ArrayList<>();  
  
    public void add(T item) {  
        list.add(item);  
    }  
  
    public T get(int index) {  
        return list.get(index);  
    }  
}
```



# Generics



```
public class Caja<T> {  
    private T contenido;  
  
    public void guardar(T contenido) {  
        this.contenido = contenido;  
    }  
  
    public T obtener() {  
        return contenido;  
    }  
}
```

```
public static void main(String[] args) {  
    Caja<Integer> cajaEnteros = new Caja<>();  
    cajaEnteros.guardar(42);  
    Integer entero = cajaEnteros.obtener();  
    System.out.println("Entero: " + entero);  
  
    Caja<String> cajaCadenas = new Caja<>();  
    cajaCadenas.guardar("Hola mundo");  
    String cadena = cajaCadenas.obtener();  
    System.out.println("Cadena: " + cadena);  
}
```



## Collections y estructuras de datos: ¿Son lo mismo?

Una colección en Java es un objeto que agrupa varios elementos en una sola unidad, lo que permite su fácil manipulación y procesamiento. Java proporciona varias interfaces de colección, como List, Set y Map, que se pueden implementar utilizando diferentes estructuras de datos para almacenar los elementos.

Por otro lado, una estructura de datos en informática es una forma de organizar y almacenar datos en un orden específico para facilitar su acceso y uso eficiente. Las estructuras de datos incluyen arrays, listas, pilas, colas, árboles, grafos, entre otras.

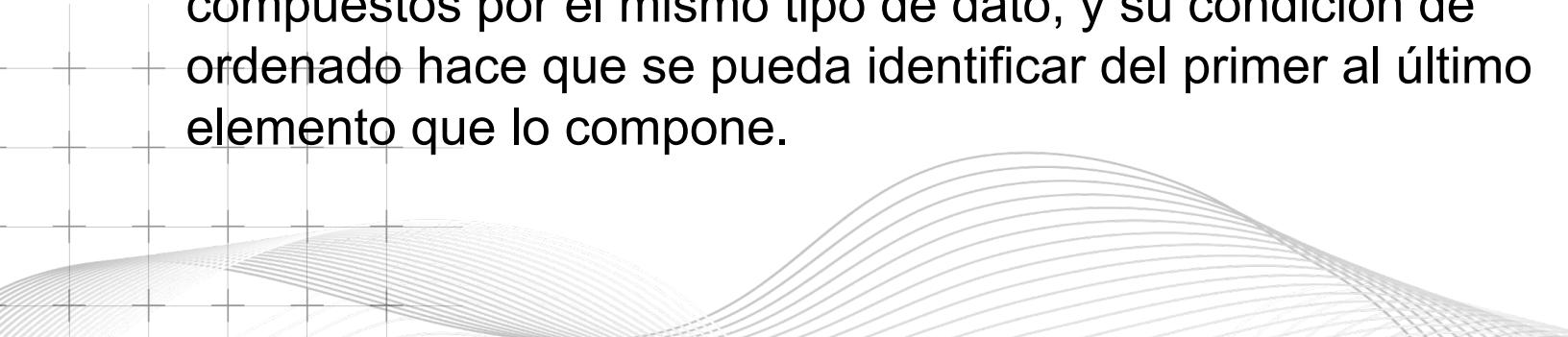




## ¿Qué es un array en programación?

Un array es un tipo de dato estructurado que permite almacenar un conjunto de datos homogéneo y ordenado, es decir, todos ellos del mismo tipo y relacionados.

Su condición de homogéneo, indica que sus elementos están compuestos por el mismo tipo de dato, y su condición de ordenado hace que se pueda identificar del primer al último elemento que lo compone.





Un array en Java es una estructura de datos que contiene una colección ordenada de elementos del mismo tipo.

Cada elemento en el array se identifica con un índice numérico, comenzando desde 0.



# Estructura de datos



## Ventajas del uso de arrays:

- Permite el acceso rápido a cualquier elemento de la estructura de datos mediante su índice.
- Es útil para almacenar y procesar grandes cantidades de datos que se relacionan entre sí.
- Los arrays son muy eficientes en términos de memoria y rendimiento.

## Desventajas del uso de arrays:

- Los arrays tienen una longitud fija y no se pueden cambiar una vez que se han creado, lo que puede limitar la flexibilidad.
- El acceso a elementos en un array es de orden  $O(1)$ , lo que significa que el tiempo de acceso es constante. Sin embargo, la inserción y eliminación de elementos en un array son operaciones costosas, ya que requieren el desplazamiento de los elementos restantes dentro del array.
- Los arrays tienen un tamaño fijo y si no se utiliza correctamente, pueden llevar a problemas de desbordamiento de memoria o pérdida de datos.





Especificando el tamaño del array al momento de la creación.

```
public class Main {  
    public static void main(String[] args) {  
        int[] numbers = new int[5];  
        numbers[0] = 3;  
        numbers[1] = 1;  
        numbers[2] = 4;  
        numbers[3] = 2;  
        numbers[4] = 5;  
        System.out.println("Números en el array: ");  
        for (int i = 0; i < numbers.length; i++) {  
            System.out.println("Número en la posición " + i + ": " + numbers[i]);  
        }  
    }  
}
```



# Arrays



Especificando los elementos del array al momento de la creación.

```
int[] numbers = {1, 2, 3, 4, 5};  
for (int i = 0; i < numbers.length; i++) {  
    System.out.println(numbers[i]);  
}
```

La clase List en Java es una interfaz que representa una estructura de datos de lista.

Una lista es una colección ordenada de elementos que se pueden acceder por un índice.

La clase List proporciona una serie de métodos para insertar, eliminar, buscar y manipular elementos en la lista.



## Ventajas del uso de listas:

- Agregar o eliminar elementos en una lista es rápido, ya que no se requiere mover los elementos restantes dentro de la lista.
- La lista se puede expandir o reducir dinámicamente según sea necesario.
- Las listas son eficientes en términos de memoria.

## Desventajas del uso de listas:

- El acceso aleatorio no es eficiente, ya que se debe recorrer la lista para encontrar un elemento específico.
- Las listas consumen más memoria que los arrays debido a los punteros de enlace adicionales.



Algunas implementaciones comunes de la interfaz List incluyen:

- ArrayList
- LinkedList
- Vector



# List - ArrayList



ArrayList es una clase de la biblioteca de Java Collections que representa una lista de objetos.

A diferencia de los arrays tradicionales en Java, ArrayList es dinámico en términos de tamaño, lo que significa que puedes agregar y eliminar elementos después de que la lista se haya creado.



# List - ArrayList



Las ventajas de usar ArrayList son:

- Pueden aumentar y disminuir de tamaño dinámicamente.
- Permiten el acceso aleatorio a los elementos, lo que significa que se puede acceder a cualquier elemento de la lista en tiempo constante.
- Proporcionan una amplia gama de métodos útiles para manipular y buscar elementos en la lista.
- Son más fáciles de usar que los arrays regulares y proporcionan una mayor flexibilidad.

Las desventajas de usar ArrayList son:

- Requieren más memoria que los arrays regulares, ya que almacenan una referencia a cada objeto en la lista, además del objeto en sí.
- El rendimiento puede ser más lento en comparación con los arrays regulares, especialmente si se está agregando o eliminando elementos frecuentemente.



# List - ArrayList



```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Creación de una lista de enteros
        ArrayList<Integer> numbers = new ArrayList<>();
        // Agregar elementos a la lista
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        // Obtener el tamaño de la lista
        System.out.println("Tamaño de la lista: " + numbers.size());

        // Acceder a un elemento en una posición específica
        System.out.println("Tercer elemento: " + numbers.get(1));

        // Eliminar un elemento
        numbers.remove(2);

        // Recorrer la lista usando un bucle for
        System.out.println("Elementos en la lista:");
        for (int i = 0; i < numbers.size(); i++) {
            System.out.println(numbers.get(i));
        }
    }
}
```



# List - ArrayList



## ¿CÓMO RECORRER UNA ARRAYLIST?

**Para recorrer por índice:** se utiliza la estructura de control clásica FOR

```
ArrayList<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");

for (int i = 0; i < items.size(); i++) {
    String item = items.get(i);
    System.out.println(item);
}
```

**Para recorrer elemento por elemento:**

se utiliza la estructura FOR EACH (por cada )

```
ArrayList<String> items = new ArrayList<>();
items.add("A");
items.add("B");
items.add("C");

for (String item : items) {
    System.out.println(item);
}
```



# Estructura For Each



El bucle for-each, también conocido como bucle "for mejorado" o "for-each loop", es una estructura de control en Java y otros lenguajes de programación que simplifica la iteración sobre colecciones de datos, como arrays, ArrayLists y otros objetos que implementan la interfaz Iterable.

La sintaxis del bucle for-each es:

```
for (tipoDeDato variable : colección) {  
    // Código a ejecutar para cada elemento  
}
```



# Ejercicios



1. Crear un ArrayList genérico que almacene objetos de una clase "Estudiante" (nombre y calificación). Utilizar un forEach para iterar a través de la lista y mostrar el nombre y la calificación de cada estudiante.
2. Implementar una clase genérica "Pareja" que tenga dos atributos de tipo genérico (T y U). Crear objetos de la clase "Pareja" que representen pares de objetos de diferentes tipos, como un objeto "String" y un objeto "Integer". Crear un ArrayList para almacenar objetos de la clase "Pareja" y utilizar un forEach para imprimir los valores de cada par.
3. Crear un método genérico que acepte un ArrayList de objetos de un tipo numérico (Integer, Double, etc.) y devuelva la suma de todos los elementos en la lista. Utilizar un forEach para iterar a través de la lista y calcular la suma.
4. Crear un ArrayList de objetos "String" y utilizar un forEach para imprimir solo aquellos elementos que tengan una longitud mayor a un valor especificado.
5. Crear un ArrayList de objetos "Empleado" (nombre y salario). Implementar un método que acepte un ArrayList de objetos "Empleado" y devuelva un nuevo ArrayList que contenga solo los empleados cuyo salario sea mayor a un valor especificado. Utilizar un forEach para iterar a través de la lista original y agregar los empleados que cumplan con la condición al nuevo ArrayList.



# List **LinkedList**

---

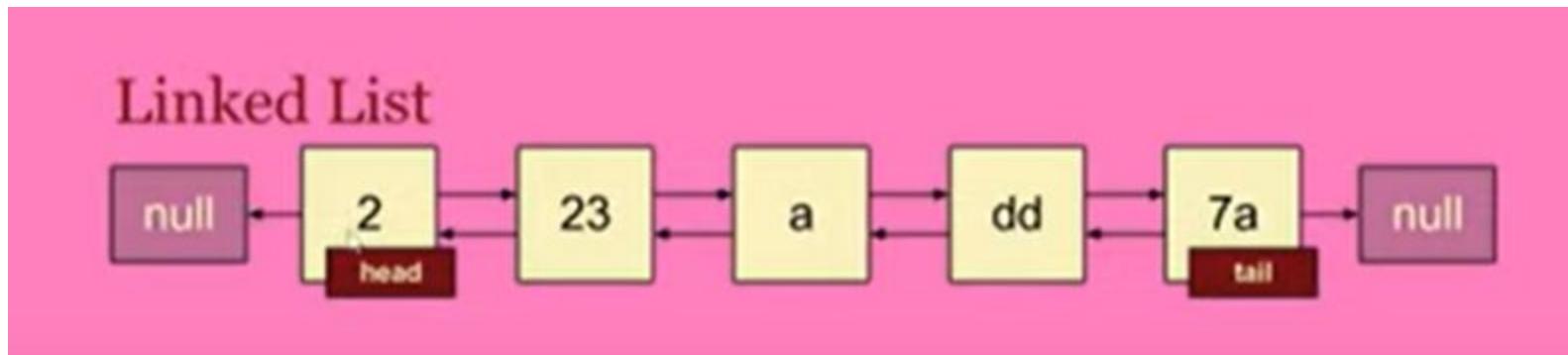


LinkedList es una clase de la biblioteca de Java Collections que representa una lista enlazada de objetos.

A diferencia de ArrayList, que almacena sus elementos en una matriz, LinkedList almacena sus elementos en una serie de nodos, donde cada nodo contiene un elemento y un enlace a otro nodo.



# List LinkedList



# List LinkedList



## Ventajas de LinkedList:

1. Inserción y eliminación eficiente: La inserción y eliminación de elementos en posiciones intermedias en un LinkedList es más eficiente que en un ArrayList, ya que solo se necesita actualizar las referencias de los elementos vecinos.
2. Implementación de Deque: LinkedList implementa la interfaz Deque, lo que permite usarla como una cola de doble extremo (double-ended queue) para agregar y eliminar elementos tanto del inicio como del final de la lista de manera eficiente.

## Desventajas de LinkedList:

1. Acceso lento a elementos: El acceso a elementos por índice en un LinkedList es más lento que en un ArrayList, ya que es necesario recorrer la lista enlazada desde el inicio o el final hasta llegar al elemento deseado.
2. Mayor uso de memoria: LinkedList utiliza más memoria que un ArrayList debido a que cada elemento necesita almacenar referencias adicionales a los elementos anterior y siguiente en la lista.



# List LinkedList



```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        // Creación de una lista de cadenas
        LinkedList<String> colors = new LinkedList<>();

        // Agregar elementos a la lista
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        // Obtener el tamaño de la lista
        System.out.println("Tamaño de la lista: " + colors.size());

        // Agregar un elemento al principio de la lista
        colors.addFirst("Orange");

        // Agregar un elemento al final de la lista
        colors.addLast("Pink");

        // Recorrer la lista usando un bucle for
        System.out.println("Elementos en la lista:");
        for (String color : colors) {
            System.out.println(color);
        }
    }
}
```



# List LinkedList



Las LinkedList en Java permiten agregar elementos fácilmente tanto al principio como al final de la lista. Puedes utilizar los métodos `addFirst()` y `addLast()` para agregar elementos al principio y al final de la lista, respectivamente. A continuación, se muestra un ejemplo en Java:

```
public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> items = new LinkedList<>();

        // Agregar elementos al final de la lista
        items.addLast("B");
        items.addLast("C");

        // Agregar un elemento al principio de la lista
        items.addFirst("A");

        // La lista ahora contiene: A -> B -> C
        for (String item : items) {
            System.out.println(item);
        }
    }
}
```



# List LinkedList



Hay otras formas de agregar elementos al principio y al final de una `LinkedList` en Java. Puedes utilizar los métodos `add(int index, E element)` y `offerFirst()` / `offerLast()` para lograrlo. A continuación, se muestra un ejemplo en Java utilizando estos métodos:

```
public class LinkedListExample {  
    public static void main(String[] args) {  
        LinkedList<String> items = new LinkedList<>();  
  
        // Agregar elementos al final de la lista  
        items.add("B");  
        items.add("C");  
  
        // Agregar un elemento al principio de la lista usando add(int index, E element)  
        items.add(0, "A");  
  
        // La lista ahora contiene: A -> B -> C  
        for (String item : items) {  
            System.out.println(item);  
        }  
  
        // Agregar otro elemento al final de la lista usando offerLast()  
        items.offerLast("D");  
  
        // Agregar otro elemento al principio de la lista usando offerFirst()  
        items.offerFirst("E");  
  
        // La lista ahora contiene: E -> A -> B -> C -> D  
        for (String item : items) {  
            System.out.println(item);  
        }  
    }  
}
```



# List LinkedList



1. **addFirst(E e)**: Este método no devuelve ningún valor (void) y lanza una excepción si no se puede agregar el elemento a la lista. En el caso de una LinkedList, dado que no hay una restricción de capacidad, no hay motivo para que se lance una excepción al agregar un elemento. Sin embargo, si estuvieras utilizando una implementación diferente de la interfaz Deque, como ArrayDeque, podría haber una restricción de capacidad y este método lanzaría una excepción si se alcanza esa capacidad.
- 2.
3. **offerFirst(E e)**: Este método devuelve un valor booleano. Devuelve `true` si se pudo agregar el elemento al principio de la lista, y `false` en caso contrario. No lanza excepciones si no se puede agregar el elemento a la lista. Al igual que con `addFirst()`, en el caso de una LinkedList, no hay restricciones de capacidad, por lo que este método siempre devolverá `true`.



# List LinkedList



## ¿CÓMO SE RECORRE UNA LINKED LIST?

Las LinkedList no tienen un acceso directo a los elementos a través de un índice como las ArrayList. Sin embargo, técnicamente aún tienen un "índice" conceptual en el sentido de que los elementos de la LinkedList tienen una posición específica en la lista y se pueden acceder secuencialmente en función de esa posición.

El bucle for-each es una forma eficiente de recorrer una LinkedList, ya que no accede a los elementos según su índice. En cambio, el bucle for-each utiliza un iterador implícitamente para recorrer los elementos de la lista de manera secuencial, lo que es más eficiente en el caso de una LinkedList.



# List LinkedList



## ¿CÓMO SE RECORRE UNA LINKED LIST?

```
// Bucle for-each
for (String elemento : linkedList) {
    System.out.println(elemento);
}

// Iterator
Iterator<String> iterador = linkedList.iterator();
while (iterador.hasNext()) {
    String elemento = iterador.next();
    System.out.println(elemento);
}
```





## ¿QUÉ ES UN ITERATOR?

El `Iterator` es una interfaz en Java que proporciona un mecanismo estandarizado para recorrer colecciones de objetos, como listas, conjuntos y mapas. El objetivo principal de un `Iterator` es permitir que los elementos de una colección sean accesados secuencialmente sin exponer su estructura interna.

Para utilizar un `Iterator`, primero debes obtenerlo a través del método `iterator()` que proporciona la colección sobre la que deseas iterar. La interfaz `Iterator` tiene tres métodos principales:

1. `hasNext()`: Este método devuelve `true` si hay más elementos en la colección para iterar, y `false` en caso contrario.
2. `next()`: Este método devuelve el siguiente elemento en la colección. Si no hay más elementos, arroja una excepción `NoSuchElementException`.
3. `remove()`: Este método elimina el último elemento devuelto por `next()` en la colección. Este método solo se puede llamar una vez por llamada a `next()` y arroja una excepción `IllegalStateException` si se llama más de una vez.

# List LinkedList



## Diferencias entre LinkedList y ArrayList:

1. Implementación: LinkedList se basa en una lista enlazada doblemente enlazada, mientras que ArrayList se basa en un array dinámico.
2. Inserción y eliminación: La inserción y eliminación de elementos en posiciones intermedias es más eficiente en LinkedList que en ArrayList.
3. Acceso a elementos: El acceso a elementos por índice es más rápido en ArrayList que en LinkedList.
4. Uso de memoria: LinkedList utiliza más memoria que ArrayList debido a las referencias adicionales que se requieren para mantener la estructura de la lista enlazada.



# EJERCICIOS



1. **Eliminar elementos pares:** Escribe un programa que, dada una LinkedList de números enteros, utilice un Iterator para eliminar todos los elementos pares de la lista.
  
2. **Invertir una lista:** Escribe un programa que, dada una LinkedList, utilice un Iterator para invertir el orden de los elementos en la lista. Intenta hacerlo sin crear una lista nueva y sin modificar directamente los enlaces entre los nodos de la LinkedList.
  
3. **Comparar dos listas:** Escribe un programa que compare dos LinkedList y verifique si contienen los mismos elementos en el mismo orden. Utiliza dos Iterator para recorrer ambas listas simultáneamente y comparar los elementos.



# LIFO -FIFO



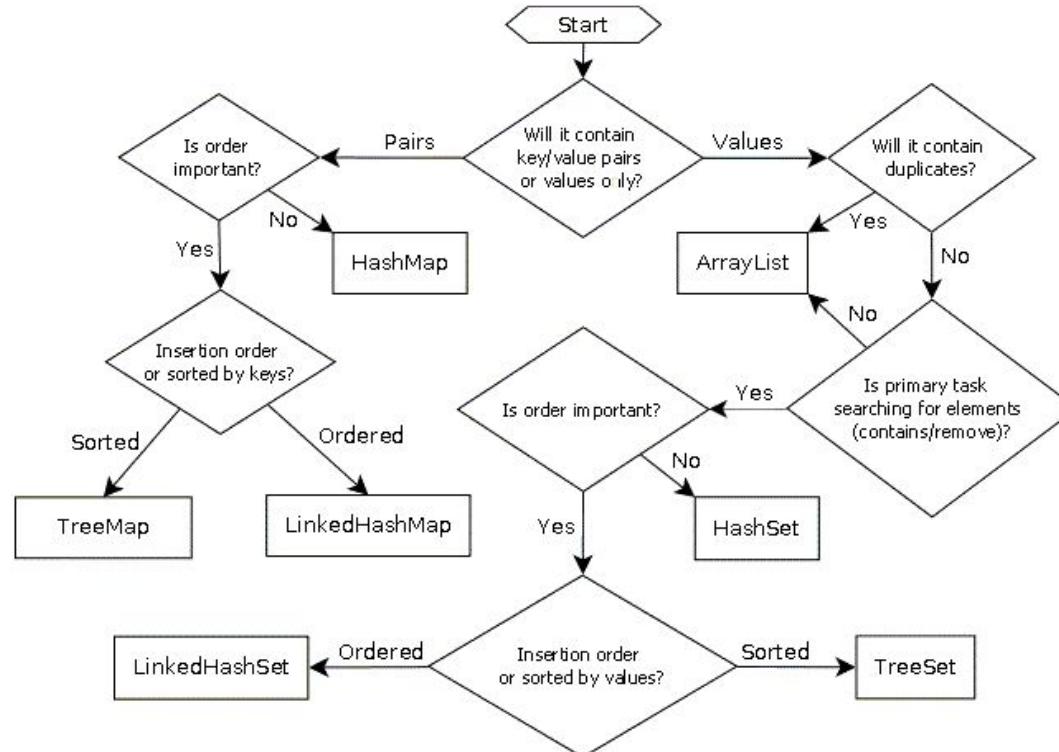
**LIFO (Last In, First Out) y FIFO (First In, First Out)** son terminologías que describen la forma en que se manejan los datos en ciertas estructuras de datos o sistemas.

**LIFO (Last In, First Out):** LIFO describe una política en la que el último elemento agregado a la estructura de datos es el primero en ser eliminado o procesado. La estructura de datos que sigue este principio es la pila (stack) y sus operaciones típicas son push (agregar) y pop (eliminar). Los elementos se agregan y se eliminan desde el "tope" de la pila, y el último elemento agregado es siempre el primero en ser eliminado.

**FIFO (First In, First Out):** FIFO describe una política en la que el primer elemento agregado a la estructura de datos es el primero en ser eliminado o procesado. La estructura de datos que sigue este principio es la cola (queue) y sus operaciones típicas son enqueue (agregar) y dequeue (eliminar). Los elementos se agregan en un extremo (cola) y se eliminan desde el otro extremo (cabeza), de modo que el primer elemento agregado es siempre el primero en ser eliminado.



# ¿QUÉ COLLECTION UTILIZAR?





# CONSULTAS?

Muchas Gracias!

