UD07 - Apuntes – Estructuras de datos en Java

Contenido

Introducción:

Las estructuras de datos son una forma especializada de organizar y almacenar datos en una computadora de manera que podamos realizar operaciones en esos datos de manera eficiente. Son fundamentales para cualquier problema de programación, ya que la elección de la estructura de datos correcta puede ser la diferencia entre un programa eficiente y uno ineficiente.

Las estructuras de datos son importantes por varias razones:

1.- **Eficiencia:** Algunas estructuras de datos son especialmente eficientes para ciertas operaciones. Por ejemplo, los arrays son muy eficientes para acceder a elementos por su índice, mientras que las listas enlazadas son eficientes para inserciones y eliminaciones.

2.- **Abstracción de datos**: Las estructuras de datos nos permiten abstraer los detalles de cómo se almacenan los datos y centrarnos en lo que queremos hacer con ellos.

3.- **Reutilización de código:** Las estructuras de datos comunes, como listas, conjuntos y mapas, se utilizan en muchos programas diferentes. Al aprender estas estructuras de datos, podemos reutilizar nuestro conocimiento y código en muchos programas.

Estructuras de datos en Java:

En Java, hay varias estructuras de datos las más comunes son:

1.- **Array:** Es la estructura de datos más básica en Java. Puede almacenar una cantidad fija de elementos del mismo tipo.
2.- **ArrayList:** Es una clase que implementa la interfaz “List”. Es similar a un array, pero su tamaño puede cambiar dinámicamente. Permite almacenar elementos duplicados y mantiene el orden de inserción.
3.- **LinkedList:** También implementa la interfaz “List”. Es una lista doblemente enlazada que permite inserciones y eliminaciones eficientes, pero el acceso a elementos individuales puede ser más lento que en un “ArrayList”.
4.- **Stack:** Es una estructura de datos de tipo LIFO (Last In, First Out). Permite agregar y quitar elementos solo desde la parte superior de la pila.
5.- **Queue**: Es una estructura de datos de tipo FIFO (First In, First Out). Los elementos se agregan al final de la cola y se eliminan desde el principio.
6.- **HashSet:** Implementa la interfaz “Set”. Almacena elementos únicos, no permite duplicados. No garantiza ningún orden específico de sus elementos.
7.- **LinkedHashSet**: Es similar a “HashSet”, pero mantiene el orden de inserción.

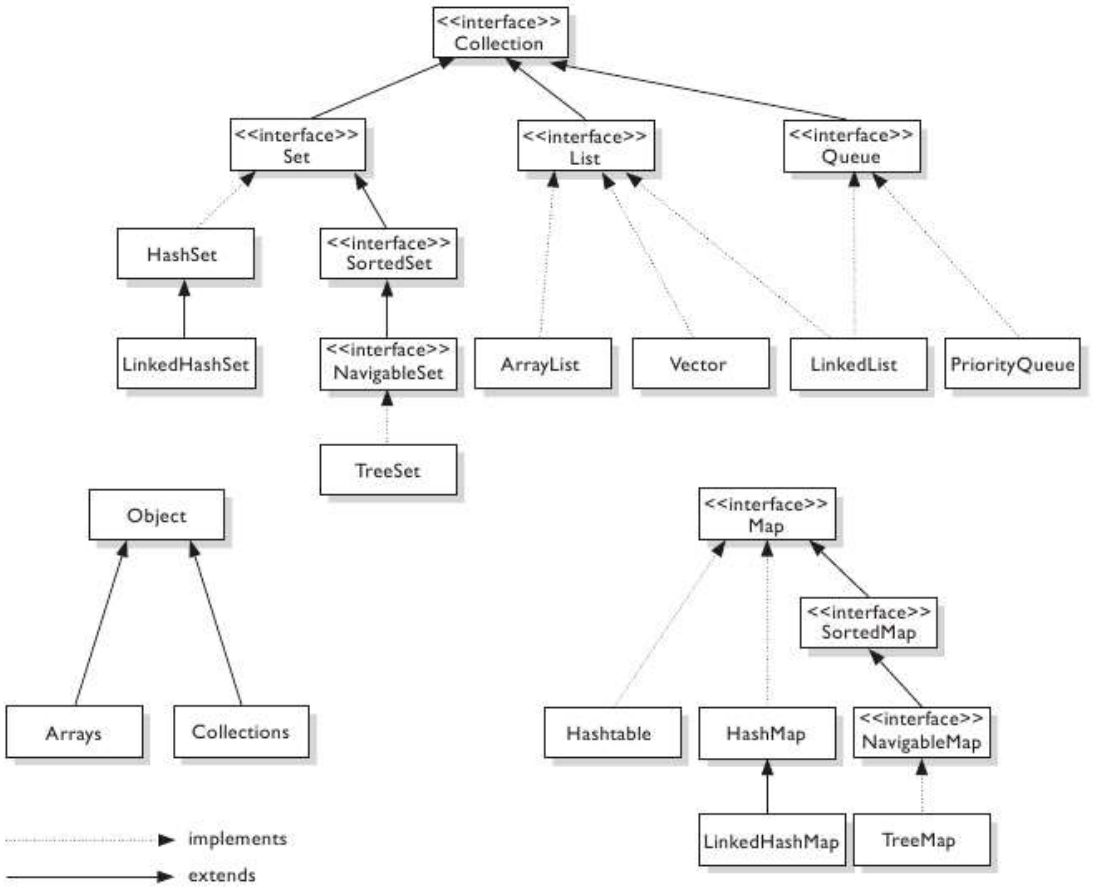
8. **TreeSet:** También implementa la interfaz “Set”. Almacena elementos únicos y los mantiene ordenados en orden ascendente.

9. **HashMap:** Implementa la interfaz “Map”. Almacena pares clave-valor. Permite valores duplicados, pero no claves duplicadas.

10. **LinkedHashMap:** Es similar a “HashMap”, pero mantiene el orden de inserción.

11. **TreeMap:** También implementa la interfaz “Map”. Almacena pares clave-valor y los mantiene ordenados por las claves.

Cada una de estas estructuras de datos tiene sus propios usos y ventajas, y es importante entender cuándo usar cada una.



LinkedList:

```
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        // Crear un LinkedList
        LinkedList<String> animales = new LinkedList<>();

        // Añadir elementos al LinkedList
        animales.add("Perro");
        animales.add("Gato");
        animales.add("Elefante");

        // Imprimir el LinkedList
        System.out.println("LinkedList: " + animales);

        // Añadir un elemento al principio
        animales.addFirst("León");
        System.out.println("Después de añadir al principio: " +
animales);

        // Añadir un elemento al final (es lo mismo que usar add())
        animales.addLast("Zebra");
        System.out.println("Después de añadir al final: " + animales);

        // Eliminar un elemento del principio
        animales.removeFirst();
        System.out.println("Después de eliminar del principio: " +
animales);
```

```
        // Eliminar un elemento del final
        animales.removeLast();
        System.out.println("Después de eliminar del final: " +
animales);

        // Acceder a elementos
        String primerElemento = animales.getFirst();
        System.out.println("Primer elemento: " + primerElemento);

        String ultimoElemento = animales.getLast();
        System.out.println("Último elemento: " + ultimoElemento);
    }
}
```

Acceso a elementos:

Puedes acceder a un elemento en una posición específica de un LinkedList en Java utilizando el método get(index), donde index es la posición del elemento que quieres obtener. Ten en cuenta que los índices en Java comienzan en 0, por lo que el primer elemento está en la posición 0, el segundo elemento está en la posición 1, y así sucesivamente.

```
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        // Crear un LinkedList
        LinkedList<String> animales = new LinkedList<>();

        // Añadir elementos al LinkedList
        animales.add("Perro");
        animales.add("Gato");
        animales.add("Elefante");
        animales.add("León");
        animales.add("Zebra");

        // Acceder a un elemento en una posición específica
        String animal = animales.get(3);
        System.out.println("El animal en la posición 3 es: " +
animal);
    }
}
```

Recorrer una LinkedList:

Para recorrer un LinkedList, puedes usar un bucle for o for-each. Aquí te dejo un ejemplo de cómo hacerlo:

```
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        // Crear un LinkedList
        LinkedList<String> animales = new LinkedList<>();

        // Añadir elementos al LinkedList
        animales.add("Perro");
        animales.add("Gato");
```

```
        animales.add("Elefante");
        animales.add("León");
        animales.add("Zebra");

        // Recorrer el LinkedList
        for (String animal : animales) {
            System.out.println(animal);
        }
    }
}
```

Stack:

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // Crear una Stack
        Stack<String> stack = new Stack<>();

        // Añadir elementos a la Stack
        stack.push("Perro");
        stack.push("Gato");
        stack.push("Elefante");

        // Imprimir la Stack
        System.out.println("Stack: " + stack);

        // Ver el elemento en la cima de la Stack sin eliminarlo
        String cima = stack.peek();
        System.out.println("Cima de la Stack: " + cima);
        System.out.println("Stack después de peek: " + stack);

        // Eliminar el elemento en la cima de la Stack
        String eliminado = stack.pop();
        System.out.println("Elemento eliminado: " + eliminado);
        System.out.println("Stack después de pop: " + stack);
    }
}
```

Por supuesto podemos recorrer un Stack usando un bucle.

Queue:

```
import java.util.Queue;
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        // Crear una Queue
        Queue<String> queue = new LinkedList<>();

        // Añadir elementos a la Queue
        queue.add("Perro");
        queue.add("Gato");
        queue.add("Elefante");

        // Imprimir la Queue
        System.out.println("Queue: " + queue);
```

```
        // Ver el elemento en el frente de la Queue sin eliminarlo
        String frente = queue.peek();
        System.out.println("Frente de la Queue: " + frente);
        System.out.println("Queue después de peek: " + queue);

        // Eliminar el elemento en el frente de la Queue
        String eliminado = queue.poll();
        System.out.println("Elemento eliminado: " + eliminado);
        System.out.println("Queue después de poll: " + queue);
    }
}
```

HashSet:

```
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        // Crear un HashSet
        HashSet<String> set = new HashSet<>();

        // Añadir elementos al HashSet
        set.add("Perro");
        set.add("Gato");
        set.add("Elefante");

        // Imprimir el HashSet
        System.out.println("HashSet: " + set);

        // Añadir un elemento duplicado
        set.add("Perro");
        System.out.println("Después de añadir un elemento duplicado: "
+ set);

        // Verificar si un elemento está en el HashSet
        boolean contienePerro = set.contains("Perro");
        System.out.println("¿Contiene 'Perro'? " + contienePerro);

        // Eliminar un elemento
        set.remove("Perro");
        System.out.println("Después de eliminar 'Perro': " + set);
    }
}
```

Si estás utilizando un HashSet con tus propias clases personalizadas, como Libro en tu ejemplo, necesitarás sobrescribir los métodos equals() y hashCode() en tu clase para que HashSet pueda evitar duplicados correctamente.

```
import java.util.HashSet;

class Libro {
    String autor;
    String titulo;

    Libro(String autor, String titulo) {
        this.autor = autor;
        this.titulo = titulo;
    }

    @Override
```

```
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Libro libro = (Libro) o;
        return autor.equals(libro.autor) &&
titulo.equals(libro.titulo);
    }

    @Override
    public int hashCode() {
        return Objects.hash(autor, titulo);
    }
}

public class Main {
    public static void main(String[] args) {
        // Crear un HashSet de Libro
        HashSet<Libro> set = new HashSet<>();

        // Añadir libros al HashSet
        set.add(new Libro("Autor1", "Titulo1"));
        set.add(new Libro("Autor2", "Titulo2"));
        set.add(new Libro("Autor3", "Titulo3"));

        // Imprimir el HashSet
        System.out.println("Número de libros en el set: " +
set.size());

        // Añadir un libro duplicado
        set.add(new Libro("Autor1", "Titulo1"));
        System.out.println("Número de libros después de añadir un
duplicado: " + set.size());
    }
}
```

LinkedHashSet:

```
import java.util.LinkedHashSet;

public class Main {
    public static void main(String[] args) {
        // Crear un LinkedHashSet
        LinkedHashSet<String> set = new LinkedHashSet<>();

        // Añadir elementos al LinkedHashSet
        set.add("Perro");
        set.add("Gato");
        set.add("Elefante");

        // Imprimir el LinkedHashSet
        System.out.println("LinkedHashSet: " + set);

        // Añadir un elemento duplicado
        set.add("Perro");
        System.out.println("Después de añadir un elemento duplicado: "
+ set);

        // Verificar si un elemento está en el LinkedHashSet
        boolean contienePerro = set.contains("Perro");
```

```
        System.out.println("¿Contiene 'Perro'? " + contienePerro);

        // Eliminar un elemento
        set.remove("Perro");
        System.out.println("Después de eliminar 'Perro': " + set);
    }
}
```

Este código hace lo mismo que el ejemplo de HashSet, pero utiliza un LinkedHashSet en lugar de un HashSet. Fíjate que el orden de los elementos en el LinkedHashSet es el mismo que el orden en que fueron insertados, a diferencia del HashSet, que no garantiza ningún orden específico.

TreeSet:

Un TreeSet en Java es una implementación de la interfaz Set que utiliza un árbol para el almacenamiento. Los elementos en un TreeSet se ordenan automáticamente en orden ascendente. Aquí tienes un ejemplo de cómo usar un TreeSet:

```
import java.util.TreeSet;

public class Main {
    public static void main(String[] args) {
        // Crear un TreeSet
        TreeSet<String> set = new TreeSet<>();

        // Añadir elementos al TreeSet
        set.add("Perro");
        set.add("Gato");
        set.add("Elefante");

        // Imprimir el TreeSet
        System.out.println("TreeSet: " + set);

        // Añadir un elemento duplicado
        set.add("Perro");
        System.out.println("Después de añadir un elemento duplicado: "
+ set);

        // Verificar si un elemento está en el TreeSet
        boolean contienePerro = set.contains("Perro");
        System.out.println("¿Contiene 'Perro'? " + contienePerro);

        // Eliminar un elemento
        set.remove("Perro");
        System.out.println("Después de eliminar 'Perro': " + set);
    }
}
```

Este código hace lo mismo que los ejemplos anteriores, pero utiliza un TreeSet en lugar de un HashSet o LinkedHashSet. Fíjate que los elementos en el TreeSet se imprimen en orden

ascendente, a diferencia de HashSet y LinkedHashSet, que no garantizan ningún orden específico.

También puede ordenar en orden descendente:

```
import java.util.TreeSet;

public class Main {
    public static void main(String[] args) {
        // Crear un TreeSet
        TreeSet<String> set = new TreeSet<>();

        // Añadir elementos al TreeSet
        set.add("Perro");
        set.add("Gato");
        set.add("Elefante");

        // Imprimir el TreeSet en orden ascendente
        System.out.println("TreeSet en orden ascendente: " + set);

        // Obtener una vista del TreeSet en orden descendente
        TreeSet<String> setDescendente = (TreeSet<String>)
set.descendingSet();

        // Imprimir el TreeSet en orden descendente
        System.out.println("TreeSet en orden descendente: " +
setDescendente);
    }
}
```

Trabajando con clases personalizadas en un TreeSet:

Para definir un orden de clasificación personalizado para una clase en un TreeSet, necesitas que tu clase implemente la interfaz Comparable y sobrescriba el método compareTo(). Veamos un ejemplo con la clase Libro:

```
import java.util.TreeSet;

class Libro implements Comparable<Libro> {
    String autor;
    String titulo;

    Libro(String autor, String titulo) {
        this.autor = autor;
        this.titulo = titulo;
    }

    @Override
    public int compareTo(Libro otro) {
        return this.titulo.compareTo(otro.titulo);
    }

    @Override
    public String toString() {
        return "Libro{" +
                "autor='" + autor + '\'' +
                ", titulo='" + titulo + '\'' +
                '}';
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        // Crear un TreeSet de Libro
        TreeSet<Libro> set = new TreeSet<>();

        // Añadir libros al TreeSet
        set.add(new Libro("Autor1", "Titulo3"));
        set.add(new Libro("Autor2", "Titulo1"));
        set.add(new Libro("Autor3", "Titulo2"));

        // Imprimir el TreeSet
        for (Libro libro : set) {
            System.out.println(libro);
        }
    }
}
```

En este código, la clase Libro implementa Comparable<Libro> y sobrescribe compareTo() para comparar libros por su título. Como resultado, cuando añadimos libros a un TreeSet, se ordenan automáticamente por su título.

HashMap:

Un HashMap en Java es una implementación de la interfaz Map que utiliza una tabla hash para el almacenamiento.

Claro, una tabla hash, también conocida como mapa hash, es una estructura de datos que implementa una matriz asociativa, un tipo de estructura que puede mapear claves a valores. Una tabla hash utiliza una función hash para calcular un índice en una matriz de cubos o ranuras, desde el cual se puede encontrar el valor deseado.

Idealmente, la función hash asignará cada clave a una ranura única. Sin embargo, en la mayoría de los casos, es posible que dos claves diferentes puedan producir el mismo índice hash. Esto se conoce como una colisión. Las tablas hash tienen estrategias para manejar estas colisiones.

En Java, `HashMap` es una clase que implementa la interfaz `Map` utilizando una tabla hash. Permite almacenar pares clave-valor donde las clales son únicas.

Es importante notar que `HashMap` en Java no mantiene ningún orden de sus elementos, ya sea el orden de inserción o el orden de las claves. Si necesitas mantener el orden, puedes usar otras clases como `LinkedHashMap` o `TreeMap`.

Veamos un ejemplo de cómo usar un HashMap:

```
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        // Crear un HashMap
        HashMap<String, String> map = new HashMap<>();
```

```
        // Añadir elementos al HashMap
        map.put("Perro", "Mamífero");
        map.put("Gato", "Mamífero");
        map.put("Elefante", "Mamífero");

        // Imprimir el HashMap
        System.out.println("HashMap: " + map);

        // Obtener un valor del HashMap
        String perroClasificacion = map.get("Perro");
        System.out.println("Perro es un: " + perroClasificacion);

        // Verificar si un elemento está en el HashMap
        boolean contienePerro = map.containsKey("Perro");
        System.out.println("¿Contiene 'Perro'? " + contienePerro);

        // Eliminar un elemento
        map.remove("Perro");
        System.out.println("Después de eliminar 'Perro': " + map);
    }
}
```

Este código crea un HashMap que mapea nombres de animales a su clasificación. Añade algunos elementos, los imprime, obtiene un valor del HashMap, verifica si un elemento está en el HashMap, y elimina un elemento.

Trabajando con clases personalizadas en un HashMap:

Si utilizas una clase personalizada como clave en un HashMap, debes sobrescribir los métodos equals() y hashCode() en tu clase. Esto es necesario para que HashMap pueda encontrar y recuperar correctamente los pares clave-valor.

El método hashCode() se utiliza para calcular un valor hash para el objeto, que HashMap utiliza internamente para encontrar la ubicación de almacenamiento (el "cubo") para esa clave. El método equals() se utiliza para comparar dos claves para la igualdad.

Veamos un ejemplo con la clase Libro:

```
import java.util.HashMap;
import java.util.Objects;

class Libro {
    String autor;
    String titulo;

    Libro(String autor, String titulo) {
        this.autor = autor;
        this.titulo = titulo;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Libro libro = (Libro) o;
        return autor.equals(libro.autor) &&
titulo.equals(libro.titulo);
    }
```

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

public class Main {
    public static void main(String[] args) {
        // Crear un HashMap de Libro a String
        HashMap<Libro, String> map = new HashMap<>();

        // Añadir libros al HashMap
        map.put(new Libro("Autor1", "Titulo1"), "Libro1");
        map.put(new Libro("Autor2", "Titulo2"), "Libro2");
        map.put(new Libro("Autor3", "Titulo3"), "Libro3");

        // Recuperar un valor del HashMap
        String libro = map.get(new Libro("Autor1", "Titulo1"));
        System.out.println(libro);  // Imprime "Libro1"
    }
}
```

En este código, la clase Libro sobrescribe equals() y hashCode() para que dos libros se consideren iguales si tienen el mismo autor y título. Como resultado, podemos usar objetos Libro como claves en un HashMap y recuperar valores correctamente.

TreeMap:

Un TreeMap en Java es una implementación de la interfaz Map que utiliza un árbol para el almacenamiento. Los elementos en un TreeMap se ordenan automáticamente en orden ascendente según las claves. Aquí tienes un ejemplo de cómo usar un TreeMap:

```
import java.util.TreeMap;

public class Main {
    public static void main(String[] args) {
        // Crear un TreeMap
        TreeMap<String, String> map = new TreeMap<>();

        // Añadir elementos al TreeMap
        map.put("Perro", "Mamífero");
        map.put("Gato", "Mamífero");
        map.put("Elefante", "Mamífero");

        // Imprimir el TreeMap
        System.out.println("TreeMap: " + map);

        // Obtener un valor del TreeMap
        String perroClasificacion = map.get("Perro");
        System.out.println("Perro es un: " + perroClasificacion);

        // Verificar si un elemento está en el TreeMap
        boolean contienePerro = map.containsKey("Perro");
        System.out.println("¿Contiene 'Perro'? " + contienePerro);

        // Eliminar un elemento
        map.remove("Perro");
```

```
        System.out.println("Después de eliminar 'Perro': " + map);
    }
}
```

Este código hace lo mismo que el ejemplo de HashMap que te mostré antes, pero utiliza un TreeMap en lugar de un HashMap. Fíjate en que las claves en el TreeMap se imprimen en orden ascendente, a diferencia del HashMap, que no garantiza ningún orden específico.

Trabajando con clases personalizadas en un TreeMap:

Si utilizas una clase personalizada como clave en un TreeMap, debes asegurarte de que tu clase implemente la interfaz Comparable y sobrescriba el método compareTo(). Esto es necesario para que TreeMap pueda ordenar las claves.

Veamos un ejemplo con la clase Libro:

```
import java.util.TreeMap;

class Libro implements Comparable<Libro> {
    String autor;
    String titulo;

    Libro(String autor, String titulo) {
        this.autor = autor;
        this.titulo = titulo;
    }

    @Override
    public int compareTo(Libro otro) {
        return this.titulo.compareTo(otro.titulo);
    }

    @Override
    public String toString() {
        return "Libro{" +
                "autor='" + autor + '\'' +
                ", titulo='" + titulo + '\'' +
                '}';
    }
}

public class Main {
    public static void main(String[] args) {
        // Crear un TreeMap de Libro a String
        TreeMap<Libro, String> map = new TreeMap<>();

        // Añadir libros al TreeMap
        map.put(new Libro("Autor1", "Titulo3"), "Libro1");
        map.put(new Libro("Autor2", "Titulo1"), "Libro2");
        map.put(new Libro("Autor3", "Titulo2"), "Libro3");

        // Imprimir el TreeMap
        for (Map.Entry<Libro, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " => " +
entry.getValue());
        }
    }
}
```

En este código, la clase Libro implementa Comparable<Libro> y sobrescribe compareTo() para comparar libros por su título. Como resultado, cuando añadimos libros a un TreeMap, se ordenan automáticamente por su título