Estructura de datos Parte 2 - Map - Set - Colas - Pilas

Map

Map es una interfaz en Java que representa una estructura de datos de asociación clave-valor.

Con un Map, puedes almacenar y recuperar pares de valores clave-valor, donde la clave es un objeto único que identifica el valor asociado.

Algunos ejemplos de uso de Map incluyen:

Almacenamiento de preferencias de usuario en una aplicación.
Almacenamiento de traducciones de texto en una aplicación multilingüe.
Almacenamiento de información de una base de datos, donde las claves son nombres de columna y los valores son los valores correspondientes.

Java proporciona varias implementaciones de la interfaz Map, incluyendo:

- HashMap
- TreeMap
- LinkedHashMap.

HashMap

Es una clase en Java que implementa la interfaz Map y almacena datos en pares clave-valor.

Utiliza una tabla hash para almacenar los elementos, lo que lo hace eficiente para la búsqueda y recuperación de valores basados en sus claves.

Una tabla hash es una estructura de datos que se utiliza para implementar un mapa (también conocido como diccionario) en la que las claves se utilizan para acceder a los valores asociados. Una tabla hash funciona transformando las claves en índices numéricos que se utilizan para acceder a los valores en una matriz o tabla.

La principal ventaja de una tabla hash es que proporciona un acceso rápido y eficiente a los elementos basados en sus claves. La mayoría de las implementaciones de tablas hash utilizan una función hash para convertir las claves en índices de la tabla, y también manejan colisiones (cuando dos claves diferentes producen el mismo índice) a través de técnicas como encadenamiento o resolución abierta.

Un ejemplo:

```
import java.util.HashMap;

public class Example {
  public static void main(String[] args) {
    // Creamos  HashMap
    HashMap<String, Integer> map = new HashMap<>();

    // Agregamos Elementos al HashMap
    map.put("John", 26);
    map.put("Jane", 30);
    map.put("Jim", 35);

    // Obtenemos el valor de una clave específica
    int age = map.get("Jane");
    System.out.println("Jane's age: " + age);

    // Chequeamos si la clave existe en HashMap
    if (map.containsKey("Jim")) {
      System.out.println("Jim is in the map");
    }
  }
}
```

TreeMap

TreeMap es una clase en Java que implementa la interfaz Map y almacena datos en una estructura de árbol ordenada.

A diferencia del HashMap, que utiliza una tabla hash y no garantiza ningún orden específico para los elementos, TreeMap garantiza que los elementos estén ordenados según su clave.

Aquí hay un ejemplo de cómo crear y usar un TreeMap en Java:

```
import java.util.TreeMap;

public class Example {
  public static void main(String[] args) {
    // Creamos TreeMap
    TreeMap<String, Integer> map = new TreeMap<>();

    // Agregamos elementos a TreeMap
    map.put("John", 26);
    map.put("Jane", 30);
    map.put("Jim", 35);
```

```
    // Obtenemos el valor de una clave específica
    int age = map.get("Jane");
    System.out.println("Jane's age: " + age);

    // Chequeamos si la clave existe en TreeMap
    if (map.containsKey("Jim")) {
      System.out.println("Jim is in the map");
    }

    // Obtenemos la primera y última clave en el TreeMap
    System.out.println("First key: " + map.firstKey());
    System.out.println("Last key: " + map.lastKey());
  }
}
```

Como puede ver, el uso de TreeMap es similar al de HashMap, pero con la garantía de que los elementos estén ordenados según la clave. Además, TreeMap también proporciona métodos para acceder al primer y último elemento en el mapa, entre otros.

LinkedHashMap

Es una clase en Java que implementa la interfaz Map y almacena los datos en pares clave-valor como HashMap. Sin embargo, a diferencia de HashMap, LinkedHashMap mantiene el orden en el que se insertaron los elementos, lo que significa que los elementos se recuperan en el orden en el que se insertaron.

Aquí hay un ejemplo de cómo crear y usar un LinkedHashMap en Java:

```
import java.util.LinkedHashMap;

public class Example {
  public static void main(String[] args) {
    // Creamos LinkedHashMap
    LinkedHashMap<String, Integer> map = new LinkedHashMap<>();

    // Agregamos elementos al LinkedHashMap
    map.put("John", 26);
    map.put("Jane", 30);
    map.put("Jim", 35);

    // Obtenemos el valor de una clave específica
    int age = map.get("Jane");
    System.out.println("Jane's age: " + age);

    // Chequeamos si la clave existe en LinkedHashMap
    if (map.containsKey("Jim")) {
      System.out.println("Jim is in the map");
```

```
    }

    // Iteramos e imprimimos los elementos
    for (String key : map.keySet()) {
      System.out.println(key + ": " + map.get(key));
    }
  }
}
```

Como se puede observar, el uso de LinkedHashMap es similar al de HashMap, pero con la garantía de que los elementos se mantendrán en el orden en el que se insertaron. Esto puede ser útil en casos donde se desea mantener un orden específico para los elementos en el mapa.

Set

Es una estructura de datos que permite almacenar elementos únicos sin repetir. Es similar a una lista, pero no permite elementos repetidos.

Hay varios tipos de objetos Set disponibles en Java, cada uno con sus propias características y funcionalidades, como:

- HashSet
- TreeSet
- LinkedHashSet

HashSet

Es una clase en Java que implementa la interfaz Set y utiliza una tabla hash para el almacenamiento.

No permite elementos duplicados y no garantiza un orden específico para los elementos.

Aquí hay un ejemplo de cómo crear y usar un HashSet en Java:

```
import java.util.HashSet;

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

    // Agregamos elementos
    set.add("John");
    set.add("Jane");
    set.add("Jim");

    // Intentamos agregar un elemento duplicado
```

```
    set.add("John");

    // Imprimimos el número de elementos
    System.out.println("Number of elements: " + set.size());

    // Chequeamos si el elemento está
    if (set.contains("Jim")) {
      System.out.println("Jim is in the set");
    }

    // Eliminamos un elemento
    set.remove("John");

    // iteramos e imprimimos
    for (String element : set) {
      System.out.println(element);
    }
  }
}
```

Como puede ver, el uso de HashSet es similar al de otros tipos de conjuntos, pero utiliza una tabla hash para el almacenamiento. Esto significa que las operaciones de búsqueda y eliminación son más rápidas en comparación con otras estructuras de datos que no utilizan una tabla hash. Sin embargo, no garantiza un orden específico para los elementos.

TreeSet

Es una clase en Java que implementa la interfaz Set y almacena los elementos en un árbol binario de búsqueda (BST, por sus siglas en inglés).

Garantiza un orden natural para los elementos, lo que significa que los elementos se ordenan en forma ascendente según su orden natural o según un comparador especificado.

Aquí hay un ejemplo de cómo crear y usar un TreeSet en Java:

```
import java.util.TreeSet;

public class Example {
  public static void main(String[] args) {
    // Creamos TreeSet
    TreeSet<Integer> set = new TreeSet<>();

    // Agregamos elementos
    set.add(3);
    set.add(1);
    set.add(2);

    // Intentamos agregar un elemento duplicado
```

```
    set.add(1);

    // Imprimimos el número de elementos
    System.out.println("Number of elements: " + set.size());

    // Chequeamos si un elemento se encuentra
    if (set.contains(2)) {
      System.out.println("2 is in the set");
    }

    // Eliminamos un elemento
    set.remove(1);

    // iteramos e imprimimos
    for (Integer element : set) {
      System.out.println(element);
    }
  }
}
```

El uso de TreeSet es similar al de otros tipos de conjuntos, pero con la garantía de que los elementos estarán ordenados. Esto puede ser útil en casos donde se desea mantener un orden específico para los elementos en el conjunto.

LinkedHashSet

Es una clase en Java que extiende la clase HashSet e implementa la interfaz Set. Utiliza una tabla hash y una lista enlazada para almacenar los elementos, permitiendo un acceso rápido a los elementos y preservando el orden en que fueron insertados.

Aquí hay un ejemplo:

```
import java.util.LinkedHashSet;

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

    // Agregamos elementos
    set.add("John");
    set.add("Jane");
    set.add("Jim");

    // Intentamos agregar un elemento duplicado
    set.add("John");
```

```
    // Imprimimos el número de elementos
    System.out.println("Number of elements: " + set.size());

    // Chequeamos si un elemento se encuentra
    if (set.contains("Jim")) {
      System.out.println("Jim is in the set");
    }

    // Eliminamos un elemento
    set.remove("John");

    // Iteramos e imprimimos
    for (String element : set) {
      System.out.println(element);
    }
  }
}
```

Como se observa, el uso de LinkedHashSet es similar al de otros tipos de conjuntos, pero también preserva el orden en que se insertaron los elementos. Esto puede ser útil en casos donde se desea mantener un orden específico para los elementos en el conjunto, pero también se requiere un acceso rápido a ellos.

Colas - Queue

Una cola es una estructura de datos que permite almacenar elementos de acuerdo a una regla de FIFO (First In, First Out), es decir, el primer elemento en ser agregado es el primero en ser removido.

Aquí hay un ejemplo de código :

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

public class Main {

	public static void main(String[] args) {
		Queue<String> queue = new LinkedList<>();
		queue.offer("A");
		queue.offer("B");
		queue.offer("C");
		queue.offer("D");
		queue.offer("E");
		System.out.println("Elementos en la cola: ");
		while (!queue.isEmpty()) {
			System.out.println(queue.poll());
		}
```

```
        }
}
```

El resultado de ejecutar este código sería:

```
Elementos en la cola:
A B C D E
```

En este ejemplo, creamos un objeto Queue de tipo LinkedList que contiene varias letras. También se muestra cómo se pueden recuperar los elementos de la cola y mostrarlos por pantalla utilizando el método poll().

Hay que tener en cuenta que cuando se utiliza una cola, los elementos se eliminan en el orden en que fueron agregados.

Pilas - Stack

Es una estructura de datos que permite almacenar elementos de acuerdo a una regla de LIFO (Last In, First Out), es decir, el último elemento en ser agregado es el primero en ser removido.

Aquí un ejemplo:

```
import java.util.Stack;

public class Main {
        public static void main(String[] args) {
                Stack<String> stack = new Stack<>();
                stack.push("A");
                stack.push("B");
                stack.push("C");
                stack.push("D");
                stack.push("E");
                System.out.println("Elementos en la pila: ");
                while (!stack.isEmpty()) {
                        System.out.println(stack.pop());
                }
        }
}
```

El resultado de ejecutar este código sería:

```
Elementos en la pila:
E D C B A
```

En este ejemplo, creamos un objeto Stack que contiene varias letras. También se muestra cómo se pueden recuperar los elementos de la pila y mostrarlos por pantalla utilizando el método pop(). Hay

que tener en cuenta que cuando se utiliza una pila, los elementos se eliminan en orden inverso al de su agregación.

Lectura requerida:
Sznajdleder, Pablo Augusto: El gran libro de Java a Fondo Editorial Marcombo.
Ceballos, Francisco Javier: Java 2. Curso de Programación, Editorial Ra-Ma

Lectura ampliatoria:
https://www.oracle.com/ar/java/