UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

LABORATORIO DE COMPUTACIÓN III

Unidad Temática N°3:
Colecciones

Material de Estudio
2° Año – 3° Cuatrimestre

Índice

INTRODUCCIÓN

En Java, las colecciones son estructuras de datos que permiten almacenar y manipular conjuntos de elementos de forma eficiente. Las colecciones son muy importantes en Java, ya que permiten manejar grandes cantidades de datos de una manera más fácil y flexible que con los arreglos.

Hay varias clases de colecciones en Java, cada una con sus propias características y usos. Los principales tipos de colecciones son:

- **Arreglos**: una estructura de datos que permite almacenar un conjunto de elementos del mismo tipo en una única variable.
- **Listas**: una colección ordenada de elementos, que permite acceder a los elementos por su índice.
- **Mapas**: una colección de pares clave-valor, que permite acceder a los valores a través de su clave.
- **Conjuntos**: una colección no ordenada de elementos únicos.

Cada tipo de colección tiene sus propias implementaciones en Java, como ArrayList, LinkedList, HashMap, TreeSet, entre otros. En general, se recomienda elegir la implementación de colección que mejor se adapte a las necesidades específicas del proyecto o aplicación en cuestión.

Arreglos (Array)

Un arreglo, también conocido como array, **es una estructura de datos que almacena una colección de elementos del mismo tipo de manera contigua en la memoria**. Los arreglos son utilizados para almacenar y manipular datos de manera eficiente y son una característica fundamental en la programación.

En Java, existen dos tipos de arreglos: unidimensionales y multidimensionales. Los arreglos unidimensionales son aquellos que contienen una sola fila de elementos, mientras que los arreglos multidimensionales pueden contener múltiples filas y columnas de elementos. Los arreglos unidimensionales se implementan utilizando corchetes `[ ]` después del tipo de datos, mientras que los arreglos multidimensionales se implementan utilizando múltiples pares de corchetes `[ ] [ ]`.

Los arreglos pueden imaginarse como una secuencia lineal de casilleros, cada uno de los cuales funciona como una variable, por tanto, el arreglo puede almacenar

tantos datos como casilleros posea. Pero como el arreglo sólo tiene un nombre, para identificar a cada casillero individual se los enumera con números naturales llamados índices. El primer índice lleva el número 0 y los siguientes con 1, 2, 3, etc. A causa de lo anterior, el último índice es equivalente al tamaño del arreglo – 1.

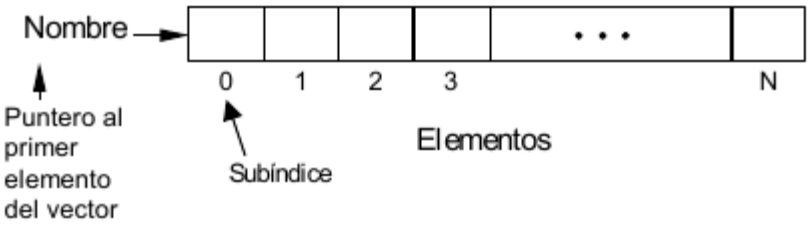

Imagen 1: Elaboración Propia.

Los arreglos poseen un tamaño estático el cual es asignado antes de utilizarlo. Tal tamaño no puede ser modificado por lo tanto debe prestarse especial atención en el momento de la creación para no seleccionar un tamaño inadecuado. Si se lo crea con una cantidad de casilleros menor a la cantidad de datos que se necesitan guardar, el arreglo se llena y algunos datos quedan fuera de la estructura. De modo contrario si se lo crea demasiado grande, por un lado se desperdicia memoria en los casilleros que no se utilicen y por otro aumenta la complejidad de los algoritmos que deben verificar continuamente cuales casilleros están ocupados y cuáles no.

Arreglos Unidimensionales

Los arreglos unidimensionales en Java son una estructura de datos que permiten almacenar múltiples valores del mismo tipo en una sola variable. Se les llama **"unidimensionales" porque los elementos del arreglo se organizan en una sola dimensión**, a lo largo de una fila. Cada elemento en el arreglo se puede acceder mediante un índice numérico que representa su posición en la fila.

La sintaxis para declarar un arreglo unidimensional en Java es la siguiente:

```
tipo[] nombreArreglo = new tipo[tamaño];
```

Imagen 2: Elaboración Propia.

Donde "***tipo***" es el tipo de dato que almacenará el arreglo (como int, double, String, etc.), "***nombreArreglo***" es el nombre que se le dará al arreglo y "***tamaño***" es la cantidad de elementos que tendrá el arreglo.

Por ejemplo, para declarar un arreglo unidimensional de enteros con 5 elementos, se podría usar la siguiente línea de código:

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

Imagen 3: Elaboración Propia.

Para asignar valores a los elementos del arreglo, se puede hacer referencia a cada elemento por su índice y asignarle un valor específico. Por ejemplo:

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

Imagen 4: Elaboración Propia.

También se pueden inicializar los elementos del arreglo en la misma línea en la que se declara el arreglo. Por ejemplo:

```
int[] numeros = {10, 20, 30, 40, 50};
```

Imagen 5: Elaboración Propia.

Para acceder a los elementos del arreglo, se puede hacer referencia a ellos por su índice numérico. Por ejemplo, para imprimir el tercer elemento del arreglo "numeros":

```
System.out.println(numeros[2]);
```

Imagen 6: Elaboración Propia.

Esto imprimirá "30" en la consola.

Arreglos Multidimensionales

Los arreglos multidimensionales son arreglos que tienen dos o más dimensiones, en lugar de una sola como los arreglos unidimensionales. En Java, se pueden declarar y usar arreglos multidimensionales de varias formas. La forma más común de declarar un arreglo bidimensional en Java es la siguiente:

```
tipo_dato[][] nombre_arreglo = new tipo_dato[num_filas][num_columnas];
```

Imagen 7: Elaboración Propia.

Donde ***tipo_dato*** es el tipo de dato que se almacenará en el arreglo, ***nombre_arreglo*** es el nombre que se le dará al arreglo, ***num_filas*** es el número de filas que tendrá el arreglo y ***num_columnas*** es el número de columnas que tendrá el arreglo. Por ejemplo, si quisieras crear un arreglo bidimensional de enteros con 3 filas y 4 columnas, se haría de la siguiente manera:

```
int[][] mi_arreglo = new int[3][4];
```

Imagen 8: Elaboración Propia.

Para acceder a un elemento específico en un arreglo bidimensional, se utilizan dos índices en lugar de uno. El primer índice se refiere a la fila y el segundo índice se refiere a la columna. Por ejemplo, para acceder al elemento en la fila 1 y columna 2 del arreglo mi_arreglo, se haría de la siguiente manera:

```
int elemento = mi_arreglo[1][2];
```

Imagen 9: Elaboración Propia.

También se pueden crear arreglos tridimensionales o con más dimensiones. La sintaxis para crear un arreglo tridimensional es la siguiente:

```
tipo_dato[][][] nombre_arreglo = new tipo_dato[num_capas][num_filas][num_columnas];
```

Imagen 10: Elaboración Propia.

Donde ***num_capas*** es el número de capas que tendrá el arreglo. Para acceder a un elemento específico en un arreglo tridimensional, se utilizan tres índices en lugar de dos. Por ejemplo, para crear un arreglo tridimensional de enteros con 2 capas, 3 filas y 4 columnas, se haría de la siguiente manera:

```
int[][][] mi_arreglo = new int[2][3][4];
```

Imagen 11: Elaboración Propia.

Para acceder a un elemento específico en el arreglo mi_arreglo, se especificarían los índices de la capa, fila y columna. Por ejemplo, para acceder al elemento en la capa 0, fila 1 y columna 2 del arreglo mi_arreglo, se haría de la siguiente manera:

```
int elemento = mi_arreglo[0][1][2];
```

Imagen 12: Elaboración Propia.

Dos ejemplos claros a los que se pueden aplicar este tipo de arreglos son el desarrollo del juego TA-TE-TI (para lo que usaremos 2 dimensiones) y el juego del cubo rubik (3 dimensiones)

Colecciones (Collection)

La interfaz Collection es una de las interfaces más importantes y ampliamente utilizadas en la biblioteca de colecciones de Java. Define el comportamiento básico que debe tener cualquier colección de objetos en Java.

En general, una colección es una estructura de datos que contiene un grupo de elementos (objetos) que pueden ser manipulados mediante una serie de operaciones. Estas operaciones incluyen agregar elementos a la colección, eliminar elementos de la colección, buscar elementos en la colección y muchas otras.

Listas (List extends Collection)

En Java, una lista es una estructura de datos que **permite almacenar y manipular elementos en secuencia**. A diferencia de los arreglos, las listas tienen tamaño dinámico y pueden crecer o disminuir según sea necesario. Además, las listas en Java pueden contener elementos de cualquier tipo de objeto.

La interfaz principal para trabajar con listas en Java es la interfaz "***List***", que se encuentra en el paquete "***java.util***". La interfaz "List" define una serie de métodos para agregar, eliminar y acceder a elementos de la lista. Algunos de los métodos más comunes son:

- **add**(elemento): agrega un elemento al final de la lista.
- **add**(index, elemento): agrega un elemento en la posición especificada.
- **remove**(index): elimina el elemento en la posición especificada.
- **get**(index): devuelve el elemento en la posición especificada.
- **size**(): devuelve el número de elementos en la lista.

Para utilizar una lista en Java, primero debemos crear una instancia de una clase que implemente la interfaz "List". Una de las clases más comunes para crear una lista es "**ArrayList**", que también se encuentra en el paquete "java.util". La sintaxis para crear una lista usando "ArrayList" es la siguiente:

```
List<String> miLista = new ArrayList<String>();
```

Imagen 13: Elaboración Propia.

En este ejemplo, estamos creando una lista de tipo "**String**" llamada "**miLista**" utilizando la clase "**ArrayList**". Podemos agregar elementos a la lista utilizando el método "**add**":

```
miLista.add("elemento1");
miLista.add("elemento2");
miLista.add("elemento3");
```

Imagen 14: Elaboración Propia.

También podemos acceder a elementos de la lista utilizando el método "**get**":

```
String elemento2 = miLista.get(1); // devuelve "elemento2"
```

Imagen 15: Elaboración Propia.

Y podemos eliminar elementos de la lista utilizando el método "remove":

```
miLista.remove(0); // elimina "elemento1" de la lista
```

Imagen 16: Elaboración Propia.

Las listas en Java son muy flexibles y se pueden utilizar en una amplia variedad de situaciones. Algunos ejemplos de uso incluyen almacenar una lista de tareas pendientes en una aplicación, almacenar una lista de productos en un catálogo en línea, o almacenar una lista de nombres de usuario y contraseñas en una base de datos.

ArrayList

ArrayList es una implementación de la interfaz List que se utiliza para almacenar y manipular datos en forma de lista. La particularidad de la implementación ArrayList es que **se basa en un arreglo subyacente que se redimensiona automáticamente a medida que se agregan o eliminan elementos de la lista**. Esto significa que las operaciones de acceso a los elementos por índice son rápidas, pero las operaciones de inserción o eliminación en posiciones intermedias de la lista pueden ser más lentas, ya que requieren la reubicación de los elementos adyacentes. **En otras palabras, si se agrega un elemento a un ArrayList que ya está lleno, se crea un nuevo arreglo más grande y se copian todos los elementos del arreglo original al nuevo arreglo. Esto se hace automáticamente en segundo plano sin que el usuario tenga que preocuparse por la redimensión manual del arreglo**.

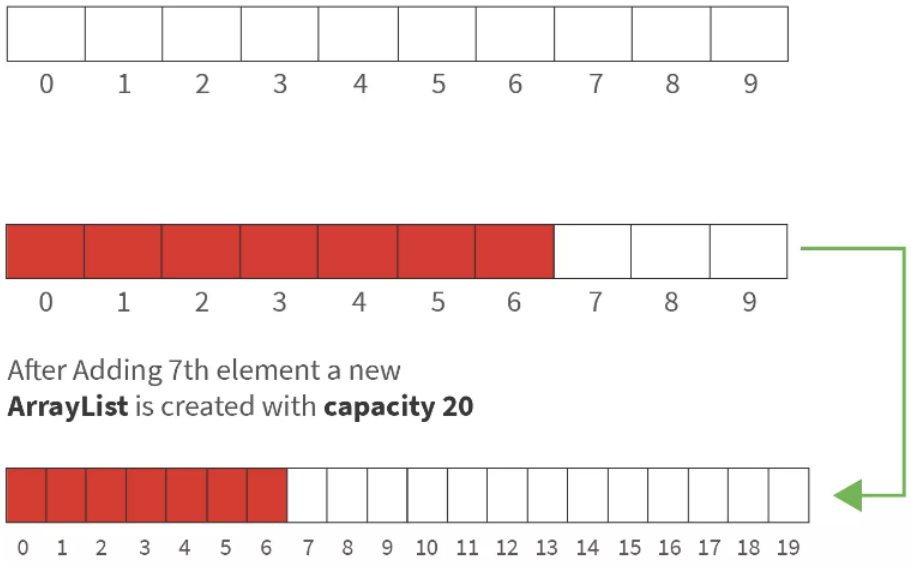

Imagen 17: Elaboración Propia.

Además, ArrayList permite el almacenamiento de elementos duplicados y admite la inserción de elementos en cualquier posición de la lista.

LinkedList

La implementación LinkedList en Java es otra forma de implementar una lista, pero a diferencia de ArrayList, **está basada en nodos enlazados**. **Cada elemento de la lista se almacena en un nodo, y cada nodo tiene un enlace que apunta al siguiente nodo en la lista**. Debido a esta estructura de nodos enlazados, la inserción y eliminación de elementos en la lista son más rápidas que en ArrayList, ya que no es necesario redimensionar el arreglo subyacente como en ArrayList.

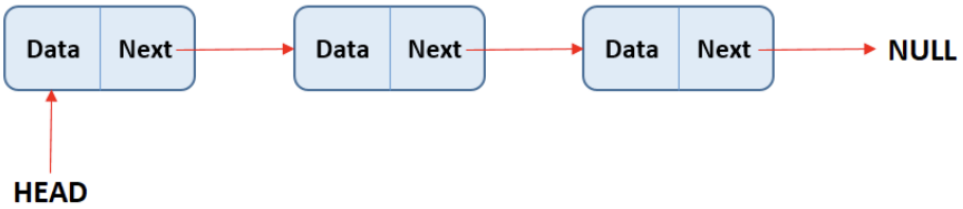

Imagen 18: Elaboración Propia.

Sin embargo, el acceso a los elementos en una LinkedList es menos eficiente que en ArrayList, ya que no se puede acceder directamente a un elemento en una posición arbitraria de la lista. En cambio, para acceder a un elemento en una LinkedList, es necesario recorrer la lista desde el principio o desde el final, dependiendo de la dirección en la que se está buscando.

En resumen, si necesitas una lista en la que se inserten y eliminen muchos elementos con frecuencia, LinkedList es una buena opción. Si necesitas una lista en la que se accedan frecuentemente los elementos por posición, entonces ArrayList es una mejor opción.

Vector

La implementación Vector en Java es similar a la de ArrayList, ya que también implementa la interfaz List y utiliza un arreglo dinámico para almacenar los elementos de la lista. Sin embargo, a **diferencia de ArrayList, Vector es sincronizado, lo que significa que es seguro para hilos de ejecución concurrentes**. Esto se debe a que Vector garantiza que ninguna operación de lectura o escritura se puede realizar en el mismo índice de la lista simultáneamente por diferentes hilos, **lo que evita la posibilidad de condiciones de carrera**.

Otra diferencia importante entre Vector y ArrayList es que Vector incrementa automáticamente su capacidad en una cantidad fija (llamada capacidad de aumento)

cuando se llena, mientras que ArrayList aumenta su capacidad en un 50% de su tamaño actual. Esto puede ser beneficioso en situaciones donde se requiere un tamaño fijo de capacidad de la lista.

Sin embargo, debido a que Vector es sincronizado, puede ser más lento que ArrayList en aplicaciones no concurrentes. Por lo tanto, en general, se recomienda utilizar ArrayList en aplicaciones no concurrentes y LinkedList si se necesita acceso frecuente a elementos en posiciones aleatorias de la lista.

Mapas (Map)

Los mapas en Java son **una colección de elementos que se almacenan en base a una clave y un valor asociado a dicha clave**. La clave es un valor único que se utiliza para acceder al valor asociado en la estructura. Los mapas son muy útiles para almacenar y recuperar información basada en una clave específica.

La interfaz Map en Java proporciona varios métodos útiles para trabajar con mapas. Algunos de los principales métodos incluyen:

- **put**(key, value): Agrega un par clave-valor al mapa.
- **get**(key): Devuelve el valor asociado a una clave dada, o null si no hay ninguna.
- **containsKey**(key): Devuelve true si el mapa contiene la clave dada, o false de lo contrario.
- **containsValue**(value): Devuelve true si el mapa contiene el valor dado, o false de lo contrario.
- **remove**(key): Elimina la entrada correspondiente a la clave dada del mapa, si existe.
- **size**(): Devuelve el número de entradas (pares clave-valor) en el mapa.
- **isEmpty**(): Devuelve true si el mapa no contiene ninguna entrada, o false de lo contrario.
- **keySet**(): Devuelve un conjunto de todas las claves en el mapa.
- **values**(): Devuelve una colección de todos los valores en el mapa.
- **entrySet**(): Devuelve un conjunto de todas las entradas (pares clave-valor) en el mapa.

Estos son solo algunos de los métodos proporcionados por la interfaz Map en Java. Dependiendo de la implementación específica de Map que se esté utilizando, puede haber otros métodos disponibles también.

HashMap

Es una implementación de la interfaz Map que **utiliza una estructura de tabla hash para almacenar pares clave-valor**. Es una de las implementaciones más comúnmente utilizadas en Java debido a su eficiencia en tiempo de ejecución y su facilidad de uso.

Una de las características clave de HashMap es que permite la inserción de valores nulos tanto en la clave como en el valor, aunque sólo puede haber una clave nula. Además, HashMap no mantiene ningún orden específico en la clave o en el valor.

La estructura de tabla hash es una estructura de datos que **utiliza una función de hash para asignar claves a una posición en la tabla**. Esto permite un acceso rápido y eficiente a los elementos almacenados, ya que el valor de la clave se utiliza para calcular la posición en la que se almacena el valor correspondiente.

En informática, una función **hash** (o simplemente hash) **es una función matemática que toma una entrada (o clave) de longitud arbitraria y devuelve una salida (o valor hash) de longitud fija**, generalmente más corta. Esta salida se utiliza a menudo como un índice en una estructura de datos (como un mapa o tabla hash) para acceder o buscar rápidamente los datos asociados a la clave.

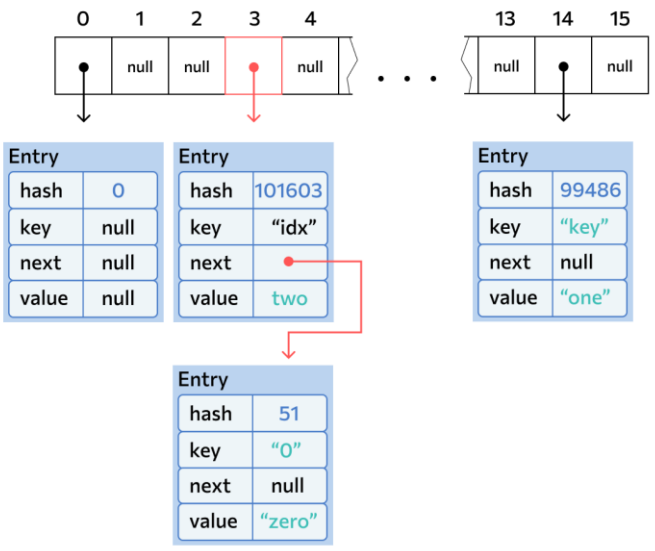

Imagen 19: Elaboración Propia.

TreeMap

La implementación TreeMap es similar a HashMap, pero en lugar de usar una tabla hash, **utiliza un árbol para almacenar los elementos en orden ascendente según las claves**. Es decir, los elementos se almacenan ordenados de acuerdo con la relación de orden natural de las claves, o un orden definido por un comparador proporcionado al construir el TreeMap. Esta es la principal diferencia con HashMap, ya que HashMap no garantiza ningún orden específico en los elementos, mientras que TreeMap los ordena según la relación de orden natural o el comparador proporcionado.

Una de las ventajas de TreeMap es que los elementos se pueden recorrer en orden de acuerdo con las claves.

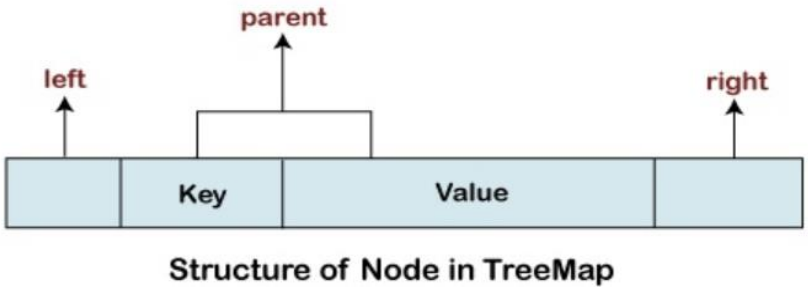

Imagen 20: Elaboración Propia.

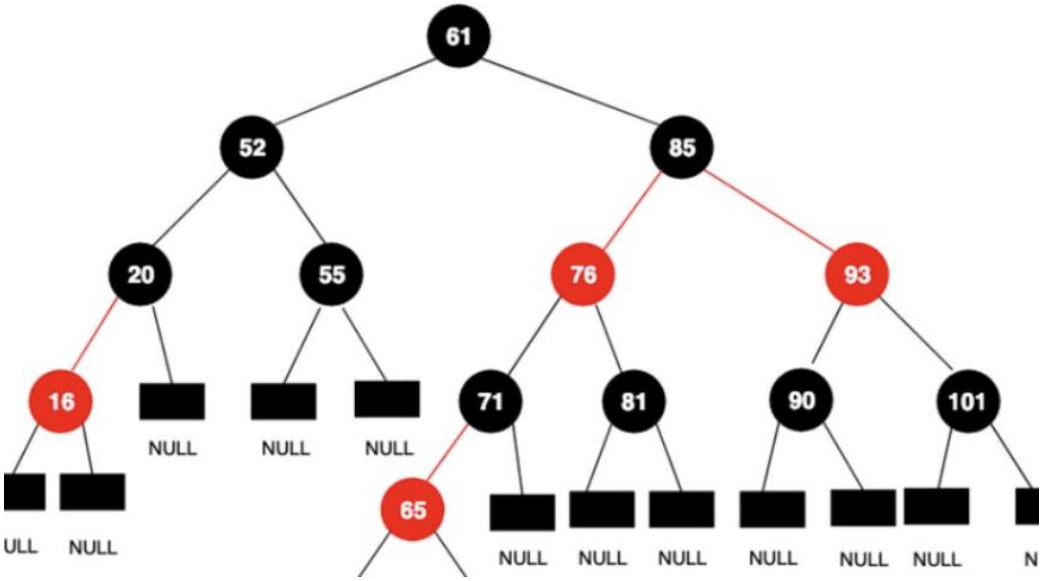

Imagen 21: Elaboración Propia.

Un caso práctico sería si se tiene un sistema de registro de notas de estudiantes y se desea ordenarlas por el nombre de los estudiantes. En este caso, se podría utilizar un TreeMap con las claves como nombres de estudiantes y los valores como las notas correspondientes. TreeMap ordenaría automáticamente las claves en orden ascendente (o descendente, según el comparador proporcionado), lo que permitiría una fácil recuperación de las notas de un estudiante específico o la generación de informes de calificaciones ordenados. Además, TreeMap también proporciona métodos para recuperar subconjuntos de claves, lo que podría ser útil para generar informes de calificaciones para grupos específicos de estudiantes.

LinkedHashMap

La implementación LinkedHashMap en Java es similar a la implementación HashMap, ya que también utiliza una tabla hash para almacenar los pares clave-valor. **La diferencia radica en que LinkedHashMap mantiene un orden de inserción**, lo que significa que los elementos se mantienen en el orden en que fueron insertados.

Además, tiene una opción para mantener un orden de acceso, lo que significa que los elementos se ordenan por la frecuencia de acceso. Cuando un elemento se accede o se agrega, se mueve al final de la lista de elementos para mantenerlo al final de la lista.

Estas particularidades hacen que LinkedHashMap sea útil en situaciones en las que es necesario mantener un orden específico de inserción o acceso a los elementos del mapa.

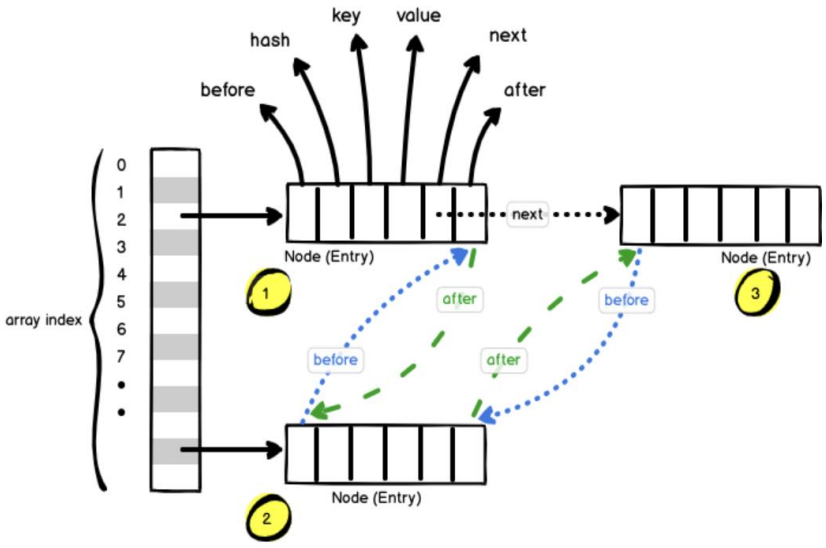

Imagen 22: Elaboración Propia.

En el caso de LinkedHashMap, la estructura de cada nodo tiene una referencia adicional llamada **before** que apunta al nodo anterior y otra llamada **after** que apunta al nodo siguiente en el orden de inserción. De esta manera, LinkedHashMap puede mantener un orden predecible basado en el momento en que se insertaron los elementos.

Un caso práctico en el que se podría utilizar la implementación de LinkedHashMap es en una aplicación de gestión de inventario en la que se necesite llevar un registro de los productos que se han agregado en orden de su ingreso.

ConcurrentHashMap

ConcurrentHashMap es una implementación que proporciona una estructura de datos en la que se pueden realizar operaciones concurrentes en diferentes segmentos de la estructura sin bloqueos. La principal ventaja de ConcurrentHashMap es que se puede acceder y modificar de manera concurrente sin necesidad de utilizar bloqueos sincronizados, lo que permite un mejor rendimiento en entornos con múltiples hilos de ejecución.

En lugar de utilizar un solo bloqueo para toda la estructura de datos, ConcurrentHashMap utiliza varios bloqueos pequeños que se aplican a diferentes segmentos de la estructura. Cada segmento se trata como un mapa separado y puede ser accedido y modificado de manera concurrente sin interferir con otros segmentos.

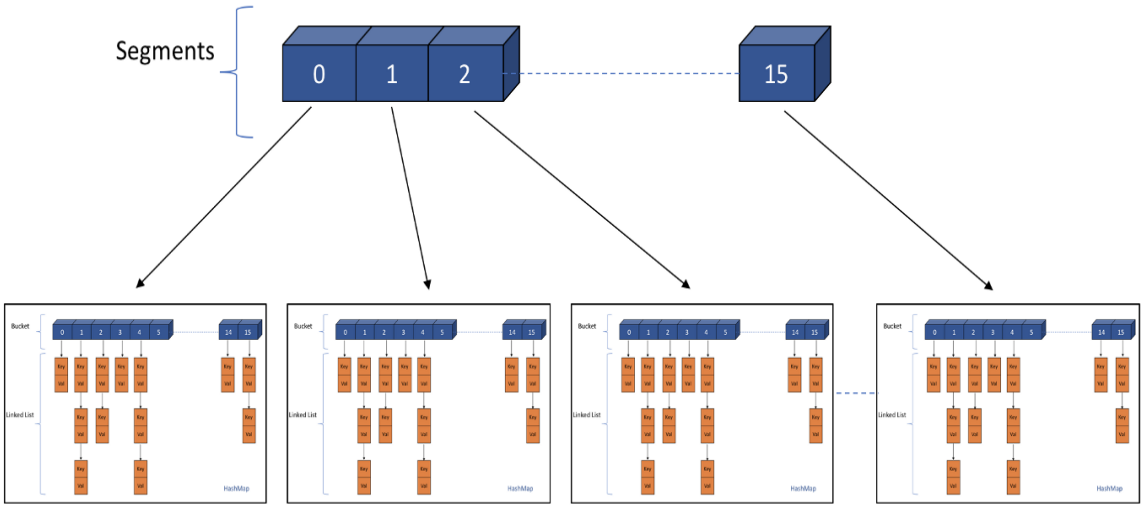

Imagen 23: Elaboración Propia.

Además de las operaciones básicas de la interfaz Map, ConcurrentHashMap proporciona una serie de métodos específicos para realizar operaciones atómicas y concurrentes, como putIfAbsent(), replace() y remove(), que permiten realizar varias operaciones atómicas en una sola llamada.

Un caso de uso común para ConcurrentHashMap es en aplicaciones de servidor que deben manejar múltiples solicitudes simultáneamente. Otro caso de uso es en aplicaciones de procesamiento de datos en tiempo real que deben procesar grandes volúmenes de datos concurrentemente.

Conjuntos (Set)

Un conjunto (Set) es una colección de elementos únicos sin un orden específico. Esto significa que no puede haber elementos duplicados en un conjunto y que los elementos no se mantienen en un orden particular como en una lista. La interfaz Set es parte del framework de colecciones de Java y es implementada por varias clases, como HashSet, TreeSet y LinkedHashSet.

Los conjuntos se utilizan comúnmente en Java para operaciones de búsqueda y eliminación eficientes de elementos, ya que las operaciones de búsqueda y eliminación son más rápidas en un conjunto que en una lista.

En Java, no se puede obtener un elemento específico de un conjunto (Set) por su índice o posición, ya que los elementos en un conjunto no están en una posición fija. En cambio, se puede utilizar el método contains() para verificar si un elemento está presente en el conjunto.

Las principales implementaciones de la interfaz Set son:

- **HashSet**: esta implementación utiliza una tabla hash para almacenar los elementos. Los elementos no se ordenan y no se garantiza un orden particular al recorrer la colección.
- **TreeSet**: esta implementación almacena los elementos en un árbol ordenado. Los elementos se ordenan de acuerdo al orden natural de los elementos o de acuerdo al orden definido por un comparador proporcionado al momento de crear el conjunto.
- **LinkedHashSet**: esta implementación es similar a HashSet, pero además mantiene el orden de inserción de los elementos en la colección.

Cada una de estas implementaciones tiene sus propias ventajas y desventajas y se deben elegir de acuerdo a las necesidades específicas de cada caso de uso.

BIBLIOGRAFÍA

- Página oficial Oracle Docs (https://docs.oracle.com/)