UT6 – PD7

**Ejercicio 1:**

En Java, el método ‘hashCode’ es esencial para las estructuras de datos como ‘HashMap’, ‘HashSet’ y ‘Hashtable’, ya que permite calcular un valor hash para los objetos. Este valor se utiliza para organizar y acceder rápidamente a los elementos almacenados. La implementación del método ‘hashCode’ varía entre las diferentes clases en Java para adaptarse a sus necesidades específicas.

Para la clase ‘Object’, que es la clase base de la cual todas las clases en Java heredan, el método ‘hashCode’ está implementado de manera muy simple. En esta implementación, el valor devuelto por ‘hashCode’ se basa típicamente en la dirección de memoria del objeto en hexadecimal. Esto significa que cada objeto tiene un valor hash único relacionado con su ubicación en la memoria. Esta implementación es eficiente en términos de rendimiento, pero no proporciona una buena dispersión de valores hash para objetos con contenido similar, ya que objetos distintos pueden tener valores hash muy cercanos o incluso iguales si la implementación subyacente del sistema lo permite.

Por otro lado, las clases ‘Integer’ y ‘String’ tienen implementaciones de ‘hashCode’ más específicas y optimizadas. En el caso de la clase ‘Integer’, el método ‘hashCode’ simplemente devuelve el valor entero mismo. Esto es posible porque los valores enteros están naturalmente bien distribuidos y no requieren procesamiento adicional para generar valores hash efectivos. Por lo tanto, si el valor del entero es 123, el ‘hashCode’ también será 123. Esta simplicidad es posible debido a la naturaleza única y directa de los números enteros.

La clase ‘String’, que representa secuencias de caracteres, utiliza una implementación más compleja para su método ‘hashCode’. El valor hash para una cadena de texto se calcula tomando en cuenta todos los caracteres de la cadena. Esta implementación asegura que incluso pequeñas diferencias en las cadenas produzcan diferentes valores hash, mejorando significativamente la dispersión y reduciendo las colisiones en las tablas hash.

La diferencia en las implementaciones del método ‘hashCode’ entre ‘Object’, ‘Integer’ y ‘String’ se debe a las diferentes características y necesidades de estos tipos de objetos. Mientras que ‘Object’ proporciona una implementación genérica basada en la dirección de memoria, ‘Integer’ aprovecha la dispersión natural de los números enteros, y ‘String’ emplea un algoritmo más sofisticado para manejar adecuadamente la dispersión de secuencias de caracteres. Estas diferencias reflejan un compromiso entre la eficiencia y la necesidad de minimizar colisiones en las estructuras de datos basadas en hash, garantizando que las operaciones como búsqueda, inserción y eliminación sean lo más eficientes posible.

**Ejercicio 2:**

**Estructura interna de un HashMap:**

Un HashMap en Java está compuesto principalmente por una tabla (arreglo) de "buckets" (cubos). Cada bucket contiene una lista enlazada (o árbol rojo-negro en Java 8 en adelante) de entradas (Entry), donde cada entrada es un par clave-valor. Aquí hay una explicación de los componentes clave:

1. Tabla (Array de Buckets): Es un arreglo de referencias a las listas de entradas. La longitud de este arreglo es una potencia de 2 (por ejemplo, 16, 32, etc.).
2. Buckets: Cada elemento del arreglo de la tabla apunta a una lista enlazada de entradas. Cuando se produce una colisión (es decir, cuando dos claves tienen el mismo valor hash), ambas entradas se almacenan en la misma lista enlazada.
3. Entrada (Entry): Cada entrada almacena una clave, un valor, el valor hash de la clave, y una referencia a la siguiente entrada en la lista enlazada.

**Inserción de Elementos**

Al insertar un elemento en el HashMap, se sigue el siguiente proceso:

1. Calcular el hash: El método hashCode de la clave se utiliza para calcular un valor hash.
2. Determinar el índice: El índice del bucket en la tabla se determina mediante el operador módulo (hash % tamaño de la tabla).
3. Insertar la entrada: Si el bucket en el índice calculado está vacío, se crea una nueva entrada y se coloca allí. Si ya hay entradas en ese bucket, se agrega la nueva entrada al inicio de la lista enlazada o se maneja de acuerdo con la política de colisión (lista enlazada o árbol rojo-negro).

**Calculo de inserción:**

Asumieremos un tamaño de la tabla de la cantidad de inserciones + el 10%, es decir, para este caso que son 4 inserciones utilizaremos una tabla de tamaño 5.

Para calcular el hash, se puede hacer de indefinidas maneras, en este caso, el cálculo será la cantidad de letras dividido 2 (división entera).

1. ‘Hola’: hash= 2
2. ‘HolaMundo’: hash= 4
3. ‘HashMap’: hash = 3
4. ‘Colecciones’: hash=5

**Índices:**

1. ‘Hola’: índice= 2%5 = 2
2. ‘HolaMundo’: índice= 4%5 = 4
3. ‘HashMap’: índice=3%5 = 3
4. ‘Colecciones’: hash=5%5=0

**Diagrama:**

|  |  |
| --- | --- |
| Índice | Contenido |
| 0 | ‘Colecciones’ (hash=0, valor = ‘Colecciones’) |
| 1 |  |
| 2 | ‘Hola’ (hash=2, valor= ‘Hola’) |
| 3 | ‘HashMap’ (hash= 3, valor= ‘HashMap’) |
| 4 | ‘HolaMundo’ (hash=4, valor = ‘HolaMundo’) |

**Ejercicio 3:**

    @Override

    public boolean equals(Object o) {

        if (this == o) return true;

        if (o == null || getClass() != o.getClass()) return false;

        Alumno alumno = (Alumno) o;

        return ID == alumno.ID && Objects.equals(fullName, alumno.fullName) && Objects.equals(email, alumno.email);

    }

    @Override

    public int hashCode() {

        return Objects.hash(ID, fullName, email);

    }