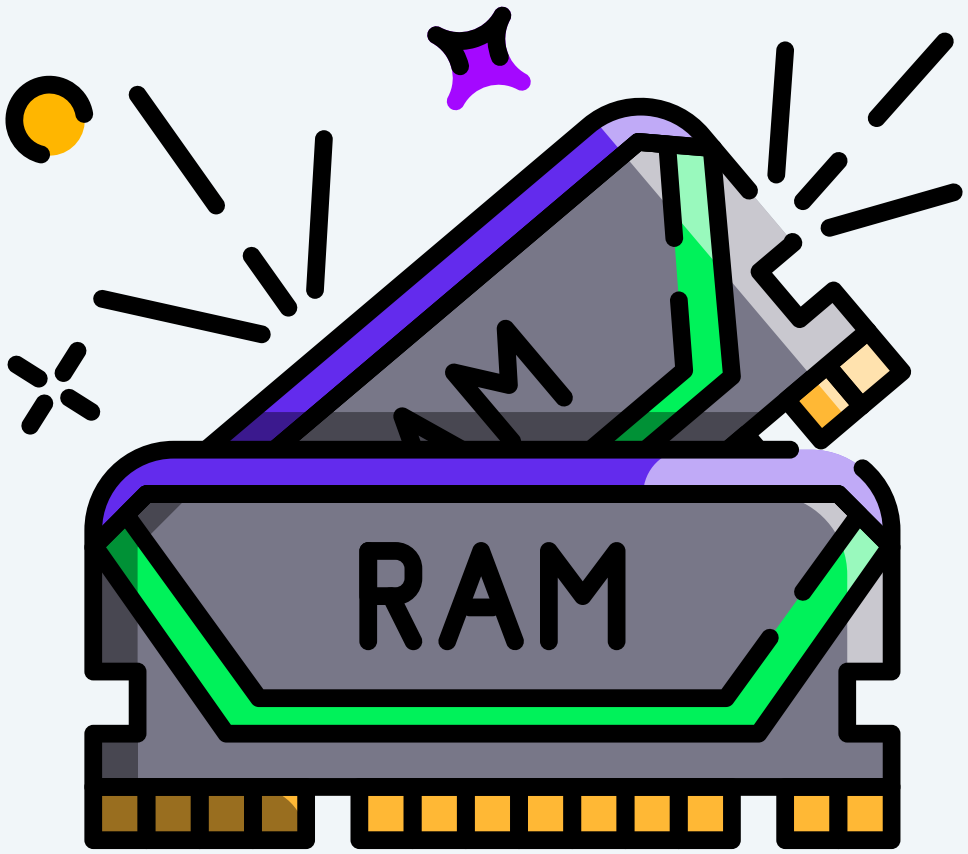
Uso de Estructuras de Datos Compactas

(optimiza y reduce el Uso de Memoria: BitSet o Boolean Array)



PABLO DEL ÁLAMO

Contexto

> Si necesitas representar un conjunto de datos donde solo se requiere un valor binario (como true/false o 0/1) para cada elemento, en lugar de usar estructuras de datos convencionales (como List o Set), puedes usar estructuras de datos compactas como un BitSet o un array de booleanos.

PABLO DEL ÁLAMO

Estas estructuras son útiles cuando se manejan grandes volúmenes de datos en los que la eficiencia de memoria es crucial.

Por ejemplo, en lugar de almacenar cada valor booleano como un Boolean independiente, puedes almacenar cada uno en un único bit, reduciendo el uso de memoria significativamente.

PABLO DEL ÁLAMO

Ejemplo: Seguimiento de Usuarios Activos en un Sistema

Imagina que tienes un sistema con miles de usuarios, y solo necesitas saber si un usuario específico está activo o inactivo.

Podrías utilizar una estructura de datos como una ArrayList o HashSet de Boolean, pero esto consumiría mucha memoria, ya que cada Boolean en Java ocupa un byte entero, mientras que un solo bit podría ser suficiente.

PABLO DEL ÁLAMO

En este caso, un BitSet es una excelente opción, ya que cada posición en el conjunto representa el estado (true o false) de un usuario y solo ocupa un bit.

PABLO DEL ÁLAMO

Código sin optimizar (usando HashSet o ArrayList de Boolean)

Este código muestra una implementación básica utilizando un HashSet para almacenar el estado de cada usuario activo.

Este enfoque consume más memoria de la necesaria, ya que el HashSet almacena cada ID completo y su estado, ocupando varios bytes por cada elemento en lugar de un solo bit.

Aunque este método es funcional, es ineficiente en términos de memoria.

PABLO DEL ÁLAMO

```java
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<Integer> activeUsers = new HashSet<>();

        // Agregar usuarios activos (usando IDs de usuario)
        activeUsers.add(1);
        activeUsers.add(50);
        activeUsers.add(300);

        // Verificar si un usuario está activo
        int userId = 50;
        boolean isActive = activeUsers.contains(userId);

        System.out.println("¿El usuario " + userId + " está activo? " + isActive);
    }
}
```

PABLO DEL ÁLAMO

Código optimizado (usando BitSet en lugar de HashSet)

A continuación, usamos BitSet para reducir el consumo de memoria.

En este caso, cada bit del BitSet representa un usuario, y el valor del bit (1 o 0) indica si el usuario está activo o inactivo.

PABLO DEL ÁLAMO

```java
import java.util.BitSet;

public class Main {
    public static void main(String[] args) {
        // Suponiendo un sistema con un máximo de 1000 usuarios
        BitSet activeUsers = new BitSet(1000);

        // Marcamos algunos usuarios como activos (estableciendo el bit en true)
        activeUsers.set(1);   // Usuario con ID 1 está activo
        activeUsers.set(50);  // Usuario con ID 50 está activo
        activeUsers.set(300); // Usuario con ID 300 está activo

        // Verificamos si un usuario específico está activo
        int userId = 50;
        boolean isActive = activeUsers.get(userId);

        System.out.println("¿El usuario " + userId + " está activo? " + isActive);
    }
}
```

PABLO DEL ÁLAMO

Explicación de la Optimización

- Uso de BitSet: Con BitSet, cada usuario es representado por un solo bit. Esto es mucho más eficiente que usar un HashSet o ArrayList, ya que solo ocupa un bit por usuario en lugar de un objeto Boolean, que en Java ocuparía 8 bits o más.

PABLO DEL ÁLAMO

- Memoria Utilizada: En lugar de almacenar cada valor como un objeto Boolean, el BitSet permite almacenar la información en bits individuales. Por ejemplo, para 1,000 usuarios, el BitSet solo consume 1,000 bits (aproximadamente 125 bytes), mientras que una ArrayList<Boolean> ocuparía mucho más espacio.

- Eficiencia en Acceso: BitSet también permite operaciones eficientes de lectura y escritura. Establecer o verificar un bit específico se realiza en tiempo constante O(1).

PABLO DEL ÁLAMO

Cuándo Usar BitSet

El BitSet es ideal cuando:

- Necesitas representar un gran conjunto de datos donde solo hay dos estados posibles (por ejemplo, activo/inactivo, presente/ausente).

- Quieres optimizar el uso de memoria en situaciones donde cada bit importa, como en sistemas embebidos o en aplicaciones que manejan datos en tiempo real.

- No necesitas valores específicos sino solo información de presencia o estado binario.

PABLO DEL ÁLAMO

BitSet en los disntintos lenguajes de programación

Aunque en esta guía hayamos ejemplificado el uso de BitSet con Java, hay equivalencias en la mayoría de principales lenguajes de programación. Te dejo una lista a continuación, para que lo pruebes en tu lenguaje de programación favorito

PABLO DEL ÁLAMO

1. Java
   - Estructura: BitSet

2. C++
   - Estructura: std::bitset y std::vector<bool>

3. Python
   - Estructura: bitarray o int con manipulación de bits

4. JavaScript
   - Estructura: Uint8Array o BigInt para manipulación de bits

5. Go
   - Estructura: math/big.Int o uint con operaciones de bits

PABLO DEL ÁLAMO

6. C# (.NET)
   - Estructura: BitArray

7. Rust
   - Estructura: bit-set (crates.io) o Vec<bool>

8. Ruby
   - Estructura: BitArray (a través de la biblioteca bitarray) o manipulación de enteros

9. Swift
   - Estructura: Set<UInt> o UInt con manipulación de bits

10. Kotlin
    - Estructura: BitSet (Java interop) o BooleanArray

11. PHP
    - Estructura: GMP o manipulación de enteros con operadores de bits

PABLO DEL ÁLAMO

¿Te ha resultado útil?

- Comparte esta guía con tu equipo o amigos desarrolladores.
- Guárdala para tenerla siempre a mano.
- ¡Dale un like o comenta si tienes preguntas!

PABLO DEL ÁLAMO