

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS ALGORITMOS Y ESTRUCTURA DE DATOS



DESCRIPCIÓN BREVE DEL CURSO En el amplio universo de la programación informática, las estructuras de datos y los algoritmos desempeñan un papel fundamental. Estos conceptos no solo son esenciales para los desarrolladores, sino que también constituyen los cimientos sobre los cuales se construyen aplicaciones eficientes y sólidas. En este artículo, exploraremos la relevancia de comprender y aplicar adecuadamente estas herramientas en el desarrollo de software. Desde la optimización del rendimiento hasta la resolución de problemas complejos, las estructuras de datos y los algoritmos conforman el núcleo de la programación moderna. Los invitamos a acompañarnos en este viaje para descubrir cómo estas piezas clave encajan en el rompecabezas de la informática.

CRISTHIAN HENRY VALENCIA

FLORES CRISTHIAN HENRY VALENCIA

FLORES 17 de junio del 2024



<< ALGORITMOS Y ESTRUCTURA DE DATOS >>
<< LABORATORIO 07:
BINARY SEARCH TREE- BST>>

CÓDIGO/DNI	APELLIDOS Y NOMBRES	FECHA
	Ortiz Puma Xiomara Mirvane Ambar	17/06/2024
2022206141	Sayritupac Asqui Jeampier	17/06/2024
2021247551	Valencia Flores Cristhian Henrry	17/06/2024

LABORATORIO 10:
HASHING - DISPERSIÓN
HASHING

Facultad de ciencias e Ingenierías Físicas y Formales.

Valencia Flores Cristhian Henrry, Sayritupac Asqui Jeampier y Ortiz Puma Xiomara Mirvane Ambar

Resumen

El hashing, o dispersión, es una técnica fundamental en la informática utilizada para mapear datos de tamaño variable a datos de tamaño fijo, típicamente a través de una función hash. Las funciones hash transforman una entrada (o clave) en una salida (o valor hash), que generalmente es una cadena de caracteres de longitud fija. Este proceso es esencial para varias aplicaciones, como la creación de tablas hash, donde las claves se asignan a índices en una matriz, lo que permite búsquedas rápidas y eficientes (Cormen et al., 2009; Knuth, 1998). Además, el hashing se utiliza en criptografía para asegurar la integridad de los datos y autenticar la información, así como en estructuras de datos y algoritmos para optimizar el rendimiento de la búsqueda, inserción y eliminación de elementos (Ferguson et al., 2010).

En el contexto de las tablas hash, uno de los principales desafíos es manejar las colisiones, que ocurren cuando dos claves distintas generan el mismo valor hash. Las técnicas de resolución de colisiones incluyen el encadenamiento, donde cada entrada de la tabla contiene una lista enlazada de todas las claves que comparten el mismo índice, y la exploración abierta, que encuentra una nueva posición en la tabla utilizando una función de sondeo. Otra técnica avanzada es el hashing perfecto, que garantiza que no ocurrirán colisiones bajo ciertas condiciones, aunque puede ser complejo de implementar (Cormen, 2011). La eficiencia de una tabla hash depende en gran medida de la calidad de la función hash y de cómo se manejen las colisiones, afectando directamente la complejidad temporal de las operaciones de búsqueda e inserción (Bentley & Sedgewick, 1997).

El hashing también tiene aplicaciones críticas en la seguridad de la información y criptografía. Las funciones hash criptográficas, como SHA-256, son diseñadas para ser unidireccionales y resistentes a colisiones, lo que significa que es computacionalmente infeasible encontrar dos entradas diferentes que produzcan el mismo valor hash. Estas propiedades hacen que las funciones hash sean esenciales para la creación de firmas digitales, la verificación de la integridad de los datos y el almacenamiento seguro de contraseñas (Joux, 2009). En el ámbito de la informática teórica y aplicada, el hashing continúa siendo una herramienta poderosa que impulsa la eficiencia y la seguridad en numerosos sistemas y aplicaciones (Ferguson et al., 2010).

Actividades

- Realice los siguientes ejercicios

1.1. Tabla hash cerrado

- Sondeo linear (Linear Probing)

- Sondeo cuadrático (Quadratic Probing)

- **Hashing doble (Double Hashing)**

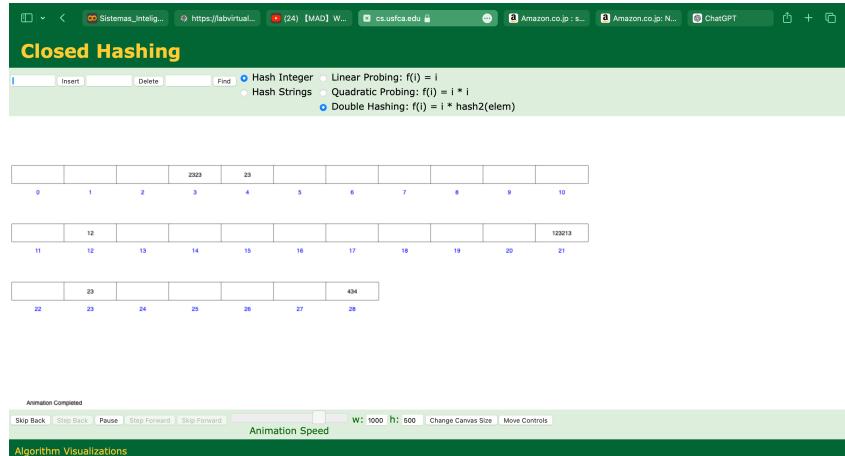
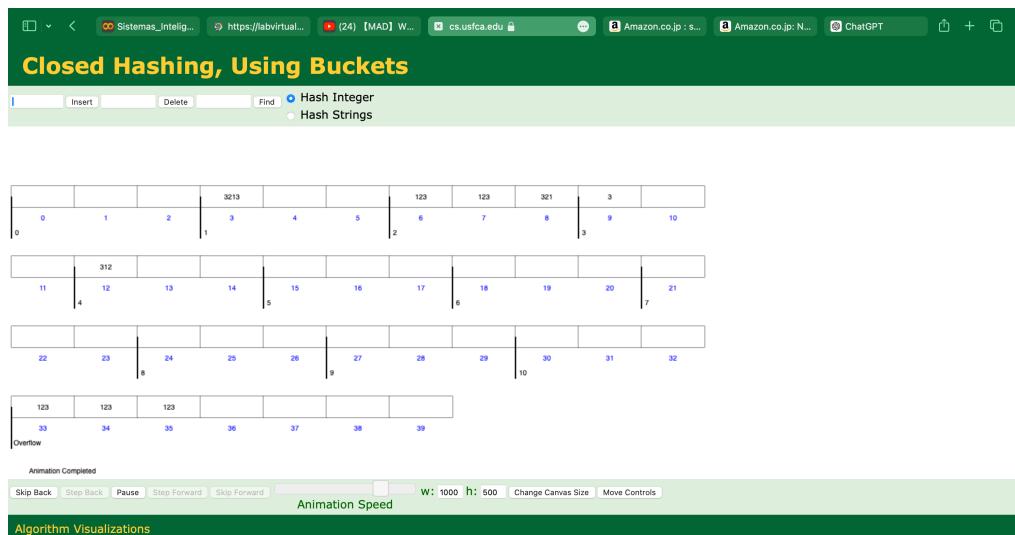


Tabla hash cerrado usando cubos



Implementando Hash cerrado

```
package Data;

public class Register <E> implements Comparable <Register<E>> {
    protected int key;
    protected E value;

    public Register(int key, E value) {
        this.key = key;
        this.value = value;
    }

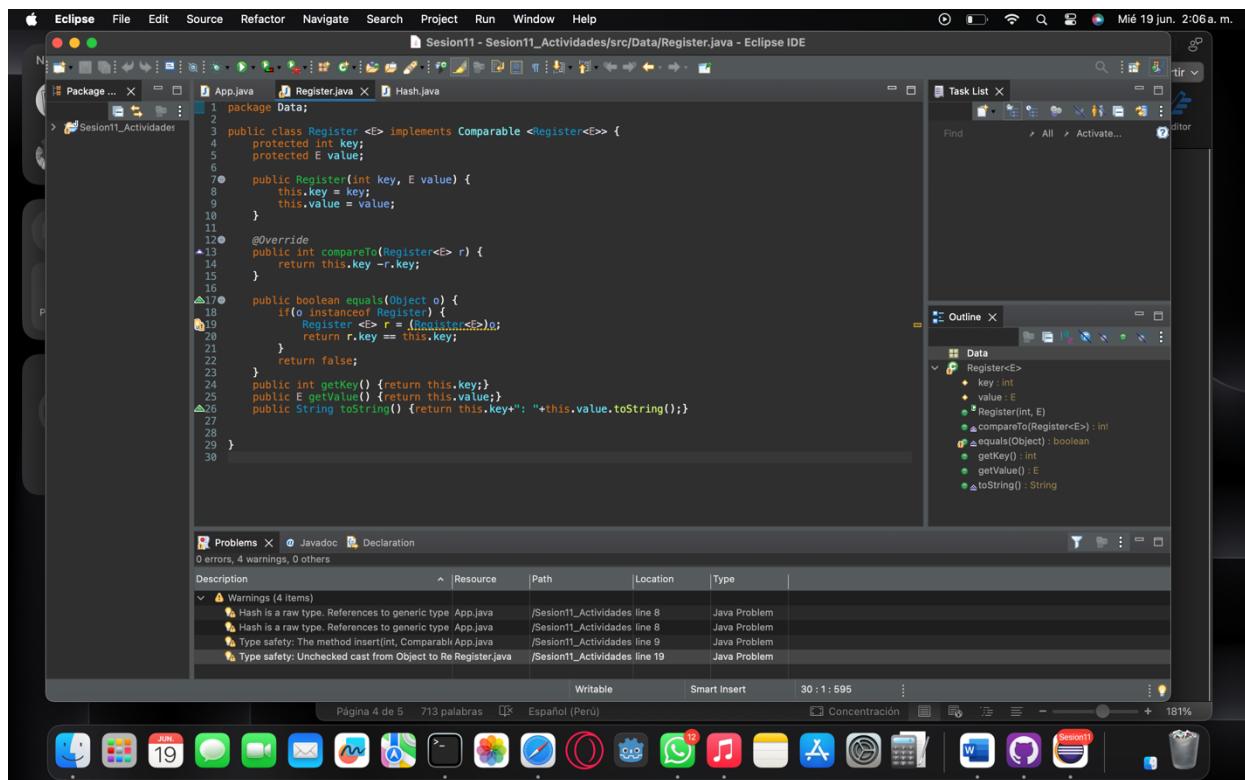
    @Override
    public int compareTo(Register<E> r) {
        return this.key - r.key;
    }

    public boolean equals(Object o) {
        if(o instanceof Register) {
            Register <E> r = (Register<E>)o;
```

```

        return r.key == this.key;
    }
    return false;
}
public int getKey() {return this.key;}
public E getValue() {return this.value;}
public String toString() {return this.key+":
"+this.value.toString();}
}

```



```

package Data;
import java.util.*;

public class Hash <E extends Comparable<E>>
{
    //Declaramos la clase elemento que sera los datos que ingresamos a
    //nuestro hash ya con su identificador
    protected class Element{
        int mark;
        Register<E> reg;
        public Element(int mark, Register<E> reg)
        {
            this.mark = mark;
            this.reg = reg;
        }
    }

    protected ArrayList<Element> table;
    protected int m;

    //Recien se inicia la clase Hash...
    //Se crea el arreglo que se usara en el hash...
    public Hash(int n)
    {
        this.m = n;
        this.table = new ArrayList<Element>(m);
    }
}

```

```

        for (int i = 0; i<m;i++)
        {
            this.table.add(new Element(0,null));
        }
    }
    private int funcionHashLineal(int key) {
        return key % m;
    }
    private int linearProbing(int dressHash, int key) {

        int i = dressHash;
        while (table.get(i).mark == 1 && table.get(i).reg.getKey() != key) {
            i = (i + 1) % m;
            if (i == dressHash) { // Si damos la vuelta completa, la tabla esta
llenada y pues se retorna -1 o null
                return -1;
            }
        }
        return i;
    }
    public void insert(int key, E reg) {
        int dressHash = funcionHashLineal(key);
        int pos = linearProbing(dressHash, key);

        if (pos != -1) {
            table.set(pos, new Element(1, new Register<E>(key, reg)));
        } else {
            System.out.println("La tabla hash esta llena, no se puede
insertar el elemento...");
        }
    }
    public E search(int key) {

        int dressHash = funcionHashLineal(key);
        int i = dressHash;

        while (table.get(i).mark != 0) {
            if (table.get(i).mark == 1 && table.get(i).reg.getKey() == key) {
                return table.get(i).reg.getValue();
            }
            i = (i + 1) % m;
            if (i == dressHash) { // Es la misma condicion en para incertar, si
esta llena no existe el dato...
                break;
            }
        }
        return null;
    }
}

```

Implementando Hash abierto (encadenamiento)

```

package Data;

public class HashA<E> {

    private static class Entry<E> {
        int key;
        E value;

        public Entry(int key, E value) {
            this.key = key;
            this.value = value;
        }
    }

    private Entry<E>[] table;

```

```

private int m; // Tamaño de la tabla
private int size; // Cantidad de elementos almacenados

public HashA(int size) {
    this.m = size;
    this.table = new Entry[m];
    this.size = 0;
}

private int hashFunction(int key) {
    return key % m;
}

public void insert(int key, E value) {
    if (size == m) {
        System.out.println("Tabla hash llena. No se puede insertar.");
        return;
    }

    int hash = hashFunction(key);

    while (table[hash] != null && table[hash].key != key) {
        hash = (hash + 1) % m; // Sonda lineal
    }

    if (table[hash] == null) {
        table[hash] = new Entry<>(key, value);
        size++;
    } else {
        table[hash].value = value; // Actualizar valor si la clave ya existe
    }
}

public E search(int key) {
    int hash = hashFunction(key);

    while (table[hash] != null) {
        if (table[hash].key == key) {
            return table[hash].value;
        }
        hash = (hash + 1) % m; // Sonda lineal
    }

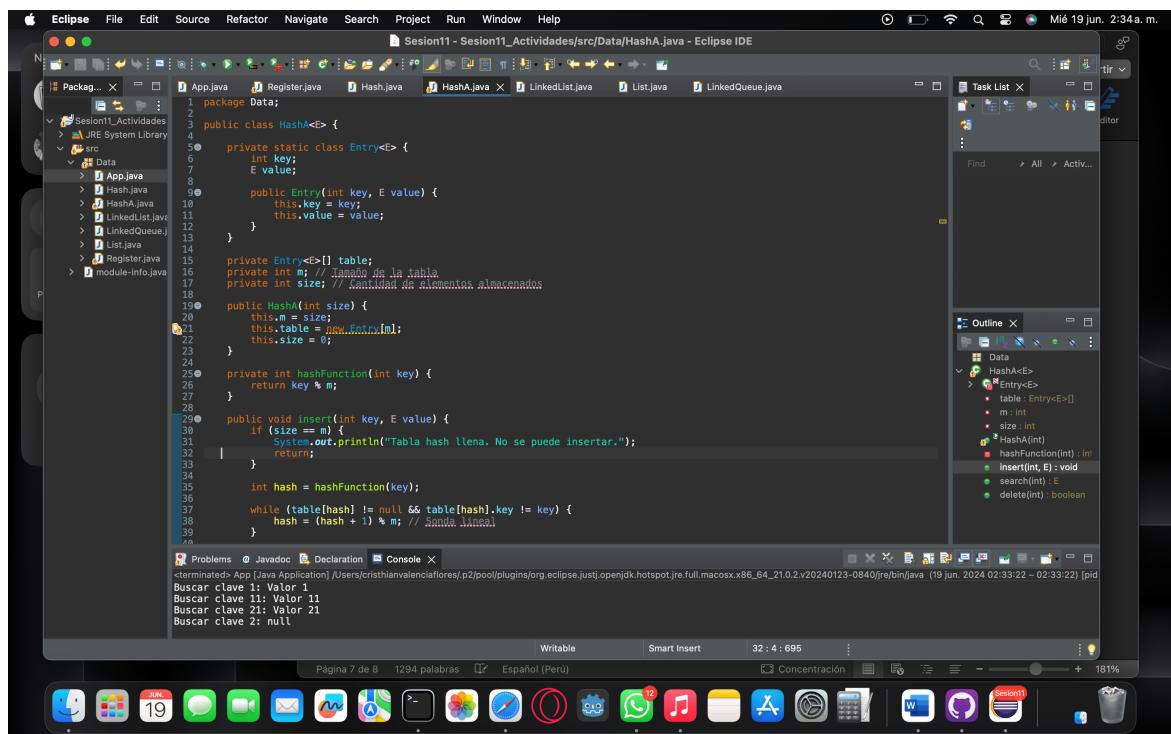
    return null; // Clave no encontrada
}

public boolean delete(int key) {
    int hash = hashFunction(key);

    while (table[hash] != null) {
        if (table[hash].key == key) {
            table[hash] = null;
            size--;
            return true; // Elemento se ha eliminado
        }
        hash = (hash + 1) % m;
    }

    return false; // Elemento no encontrado
}

```



```

Problems Javadoc Declaration Console X
<terminated> App [Java Application] /Users/cristianvalenciaflores/p2pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_21.0.2.v20240123-0840/jre/bin/java (19 jun. 2024 02:33:22 - 02:33:22) [pid
Buscar clave 1: Valor 1
Buscar clave 11: Valor 11
Buscar clave 21: Valor 21
Buscar clave 2: null

```

Ejercicios

1. Implementando funciones hash

i. Método del cuadrado

```

// Método del cuadrado para la función hash
private int hashSquare(int key) {
    int square = key * key;
    String squareString = String.valueOf(square);
    int midIndex = squareString.length() / 2;
    String hashString = squareString.substring(midIndex - 1, midIndex + 1);
    return Integer.parseInt(hashString) % m;
}

```

```

76 // Método del cuadrado para la función hash
77 private int hashSquare(int key) {
78     int square = key * key;
79     String squareString = String.valueOf(square);
80     int midIndex = squareString.length() / 2;
81     String hashString = squareString.substring(midIndex - 1, midIndex + 1);
82     return Integer.parseInt(hashString) % m;
83 }

```

ii. Método por pliegue aplicando suma

```

private int hashFoldingSum(int key) {
    String keyString = String.valueOf(key);
    int sum = 0;

    // Suma los dígitos de la clave
    for (int i = 0; i < keyString.length(); i++) {

```

```

        sum += Character.getNumericValue(keyString.charAt(i));
    }

    return sum % m;
}

87  private int hashFoldingSum(int key) {
88      String keyString = String.valueOf(key);
89      int sum = 0;
90
91      // Suma los dígitos de la clave
92
93      for (int i = 0; i < keyString.length(); i++) {
94          sum += Character.getNumericValue(keyString.charAt(i));
95      }
96
97      return sum % m;
98  }

```

- Estas clases han sido implementadas en forma privada pues son funciones que solo necesitan ser accedidas por el programa mas no por el usuario común.

Dispersando un archivo con Hash Cerrado

```

package Data;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class HashEmpleado {

    private Empleado[] tabla;
    private int m; // Tamaño de la tabla hash
    private int n; // Cantidad de empleados

    public HashEmpleado(int m) {
        this.m = m;
        this.tabla = new Empleado[m];
        this.n = 0;
    }

    private int hash(int codigo) {
        return codigo % m;
    }

    private int buscarPosicion(int codigo) {
        int hashInicial = hash(codigo);
        int hash = hashInicial;
        int k = 0;
        while (tabla[hash] != null && tabla[hash].getCodigo() != codigo) {
            k++;
            hash = (hashInicial + k * k) % m;
        }
        return hash;
    }

    public void insertar(Empleado empleado) {
        if (n == m) {
            System.out.println("Tabla hash llena. No se puede insertar más
elementos.");
            return;
        }
        int posicion = buscarPosicion(empleado.getCodigo());
        tabla[posicion] = empleado;
        n++;
    }

    public void mostrarTabla() {
        System.out.println("Tabla Hash de Empleados:");
    }
}

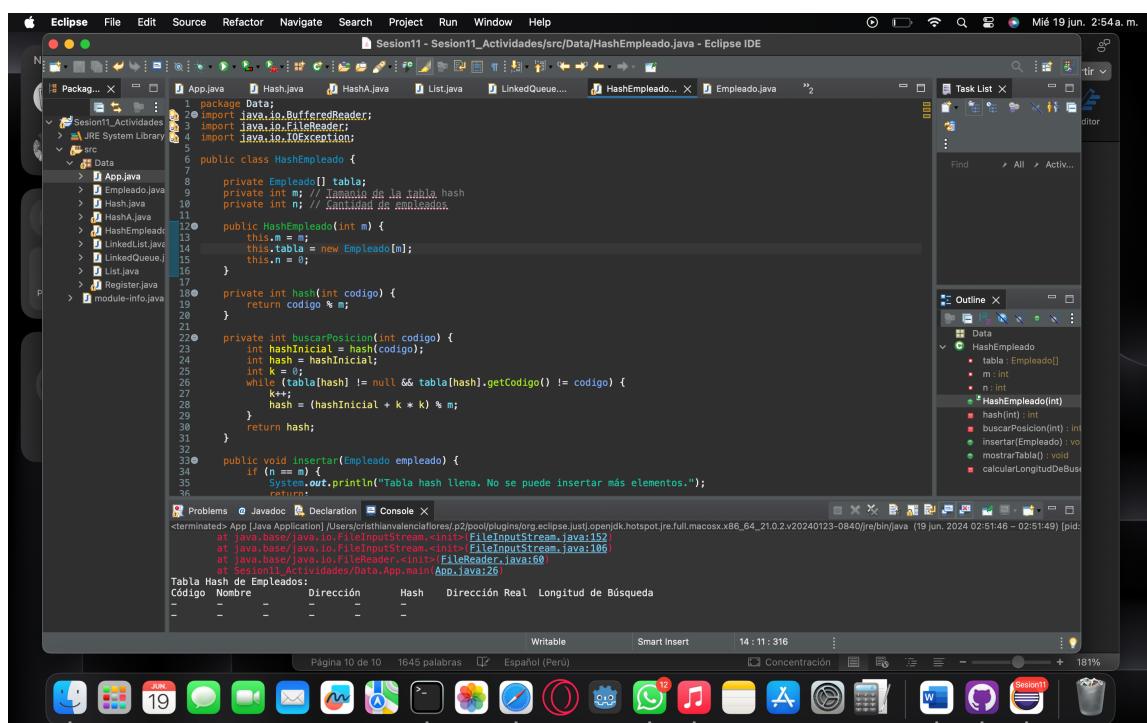
```

```

        System.out.println("Código\tNombre\t\tDirección\tHash\tDirección Real\tLongitud de Búsqueda");
        for (int i = 0; i < m; i++) {
            if (tabla[i] != null) {
                int hash = hash(tabla[i].getCodigo());
                System.out.printf("%d\t%s\t%s\t%d\t%d\t%d\n",
                    tabla[i].getCodigo(), tabla[i].getNombre(),
                    tabla[i].getDireccion(), hash, i,
                    calcularLongitudDeBusqueda(tabla[i].getCodigo())));
            } else {
                System.out.printf("-\t-\t-\t-\t-\t-\n");
            }
        }
    }

    private int calcularLongitudDeBusqueda(int codigo) {
        int hashInicial = hash(codigo);
        int hash = hashInicial;
        int k = 0;
        while (tabla[hash] != null && tabla[hash].getCodigo() != codigo) {
            k++;
            hash = (hashInicial + k * k) % m;
        }
        return k + 1; // Se agrega 1 porque se cuenta el primer intento tambien
    }
}

```



Funciones adicionales para la lectura del archivo

```

private static int calcularMDesdeArchivo(String nombreArchivo) {
    int cantidadElementos = 0;
    try (BufferedReader br = new BufferedReader(new
FileReader(nombreArchivo))) {
        String primeraLinea = br.readLine();
        cantidadElementos = Integer.parseInt(primeraLinea.trim());
    } catch (IOException e) {
        e.printStackTrace();
    }
    // Calcular m como un primo cercano a la cantidad de elementos más un
    // 40% adicional
    int cantidadCon40PorCientoAdicional = (int) (cantidadElementos * 1.4);
    return encontrarPrimoCercano(cantidadCon40PorCientoAdicional);
}

```

```

    }

    private static int encontrarPrimoCercano(int numero) {
        while (!esPrimo(numero)) {
            numero++;
        }
        return numero;
    }

    private static boolean esPrimo(int numero) {
        if (numero <= 1) {
            return false;
        }
        if (numero <= 3) {
            return true;
        }
        if (numero % 2 == 0 || numero % 3 == 0) {
            return false;
        }
        int i = 5;
        while (i * i <= numero) {
            if (numero % i == 0 || numero % (i + 2) == 0) {
                return false;
            }
            i += 6;
        }
        return true;
    }
}

```

Resultado:
(Solo estructura)

Código	Nombre	Dirección	Hash	Dirección Real	Longitud de Búsqueda
-	-	-	-	-	-
-	-	-	-	-	-

3. Dispersando un archivo con Hash Abierto

```

package Data;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class HashAbierto {
    private ListaEnlazada[] tabla;
    private int m; // Tamaño de la tabla hash

    public HashAbierto(int m) {
        this.m = m;
        this.tabla = new ListaEnlazada[m];
        for (int i = 0; i < m; i++) {
            tabla[i] = new ListaEnlazada();
        }
    }

    private int hash(int codigo) {
        return codigo % m;
    }

    public void insertar(Empleado empleado) {
        int indice = hash(empleado.getCodigo());
        tabla[indice].agregar(empleado);
    }
}

```

```
}

public Empleado buscar(int codigo) {
    int indice = hash(codigo);
    return tabla[indice].buscar(codigo);
}

public void mostrarTabla() {
    for (int i = 0; i < m; i++) {
        System.out.printf("Lista en posición %d:\n", i);
        tabla[i].mostrar();
        System.out.println();
    }
}

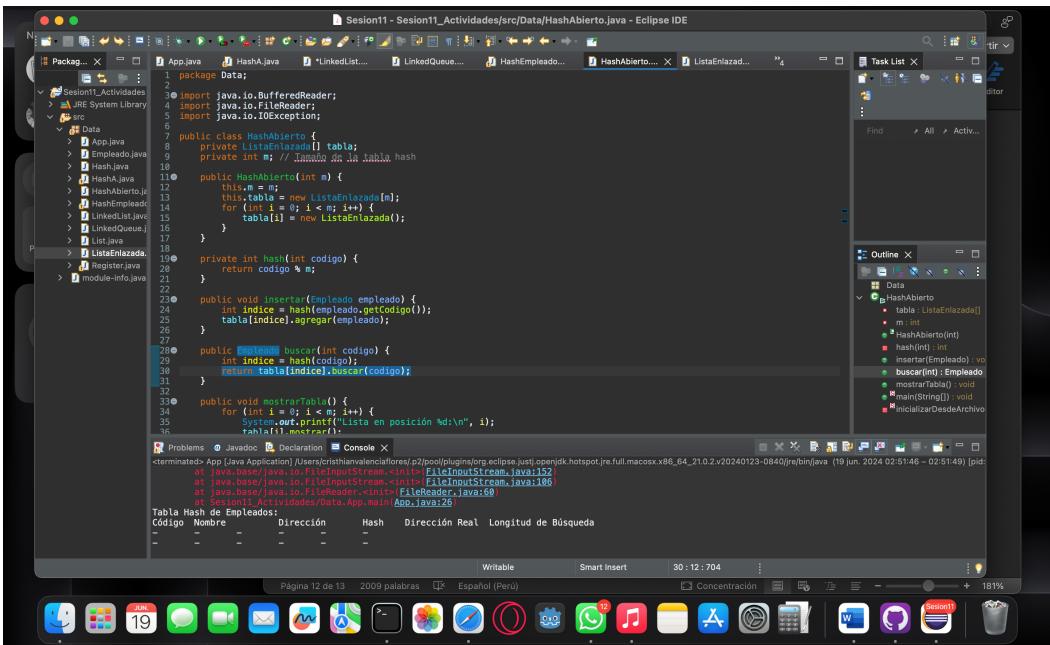
public static void main(String[] args) {
    HashAbierto hashAbierto = inicializarDesdeArchivo("EMPLEADO.TXT");

    System.out.println("Tabla Hash con Encadenamiento:");
    hashAbierto.mostrarTabla();
}

private static HashAbierto inicializarDesdeArchivo(String nombreArchivo) {
    int cantidadRegistros = 0;
    HashAbierto hashAbierto = null;
    try (BufferedReader br = new BufferedReader(new
FileReader(nombreArchivo))) {
        String primeraLinea = br.readLine();
        cantidadRegistros = Integer.parseInt(primeraLinea.trim());

        hashAbierto = new HashAbierto(cantidadRegistros);

        String linea;
        while ((linea = br.readLine()) != null) {
            String[] partes = linea.split(",");
            int codigo = Integer.parseInt(partes[0].trim());
            String nombre = partes[1].trim();
            String direccion = partes[2].trim();
            Empleado empleado = new Empleado(codigo, nombre, direccion);
            hashAbierto.insertar(empleado);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return hashAbierto;
}
}
```



Contar Frecuencia de Palabras usando Tabla Hash

```
package Data;

public class TablaHashFrecuencia {
    private int tamaño;
    private ListaEnlazada[] tabla;

    public TablaHashFrecuencia(int tamaño) {
        this.tamaño = tamaño;
        this.tabla = new ListaEnlazada[tamaño];
        for (int i = 0; i < tamaño; i++) {
            tabla[i] = new ListaEnlazada();
        }
    }

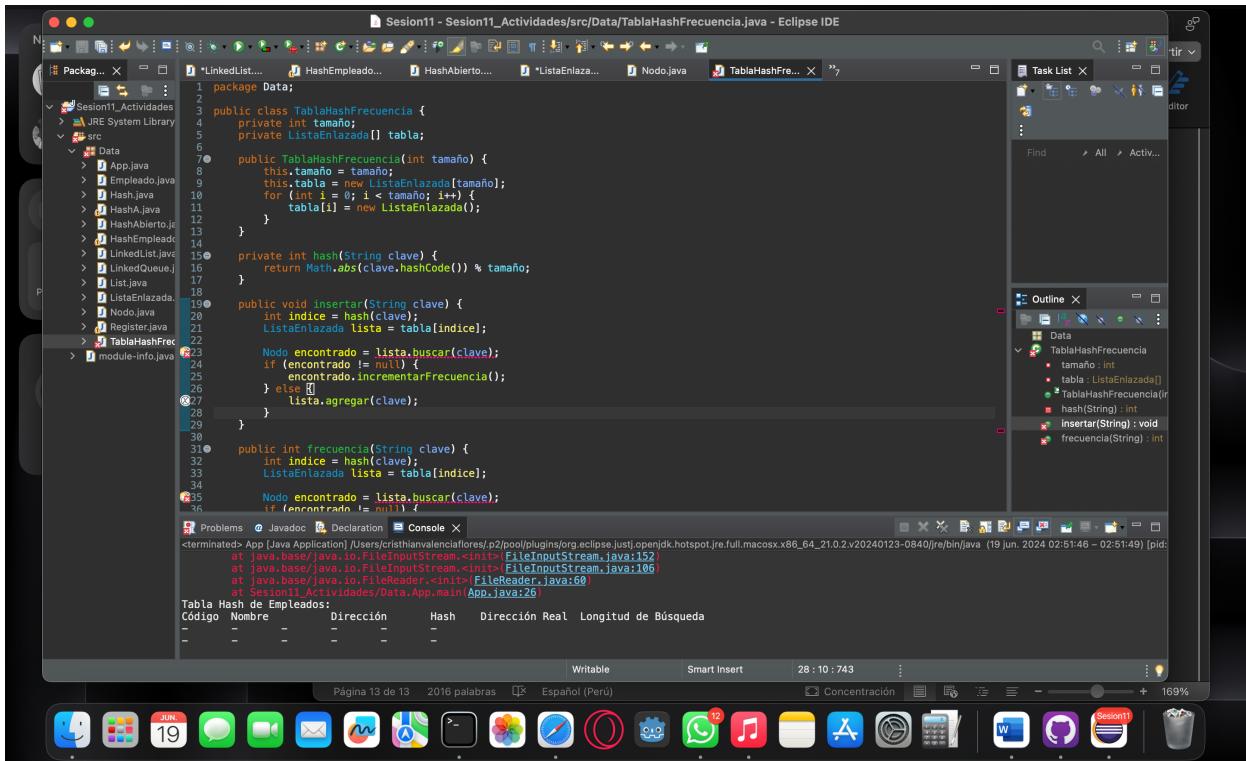
    private int hash(String clave) {
        return Math.abs(clave.hashCode()) % tamaño;
    }

    public void insertar(String clave) {
        int indice = hash(clave);
        ListaEnlazada lista = tabla[indice];

        Nodo encontrado = lista.buscar(clave);
        if (encontrado != null) {
            encontrado.incrementarFrecuencia();
        } else {
            lista.agregar(clave);
        }
    }

    public int frecuencia(String clave) {
        int indice = hash(clave);
        ListaEnlazada lista = tabla[indice];

        Nodo encontrado = lista.buscar(clave);
        if (encontrado != null) {
            return encontrado.getFrecuencia();
        } else {
            return 0;
        }
    }
}
```



Explicacion:

Esta clase implementa una tabla hash para contabilizar la frecuencia de aparición de palabras o claves. Utiliza una función de hash para determinar el índice de cada clave en la tabla, y maneja colisiones mediante el uso de listas en cada bucket. El método insertar incrementa la frecuencia de una clave si ya está presente o la añade con frecuencia inicial 1 si no lo está. frecuencia devuelve la frecuencia actual de una clave buscada. Esta estructura ofrece una forma eficiente de mantener y consultar la frecuencia de elementos en un conjunto de datos.

Implementar una Tabla Hash para Encontrar Pares que Suman un Dado Valor

```

package Data;

public class TablaHashSuma {
    private int tamaño;
    private int[][] tabla;

    public TablaHashSuma(int tamaño) {
        this.tamaño = tamaño;
        this.tabla = new int[tamaño][];
    }

    private int hash(int clave) {
        return clave % tamaño;
    }

    public void insertar(int clave) {
        int indice = hash(clave);
        if (tabla[indice] == null) {
            tabla[indice] = new int[]{clave};
        } else {
            int[] bucket = tabla[indice];
            int[] newBucket = new int[bucket.length + 1];
            System.arraycopy(bucket, 0, newBucket, 0, bucket.length);
            newBucket[bucket.length] = clave;
            tabla[indice] = newBucket;
        }
    }
}

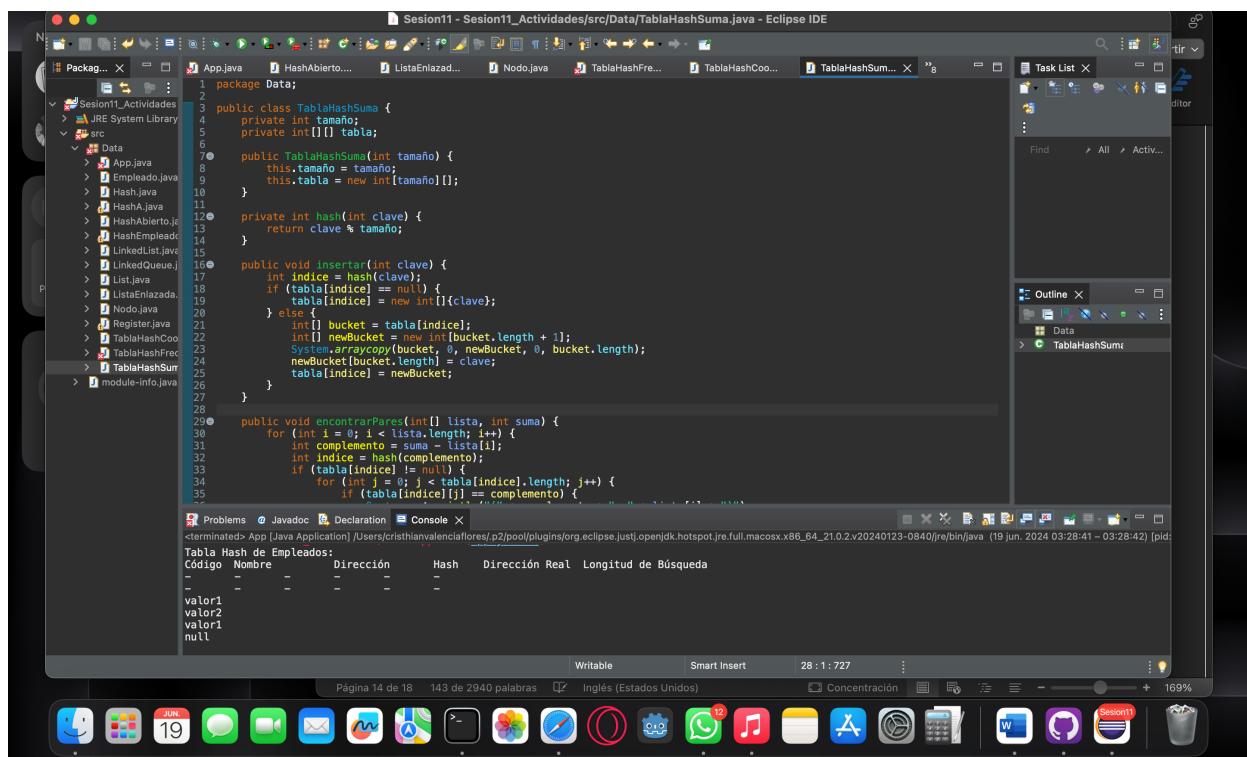
```

```

    }

    public void encontrarPares(int[] lista, int suma) {
        for (int i = 0; i < lista.length; i++) {
            int complemento = suma - lista[i];
            int indice = hash(complemento);
            if (tabla[indice] != null) {
                for (int j = 0; j < tabla[indice].length; j++) {
                    if (tabla[indice][j] == complemento) {
                        System.out.println("(" + complemento + ", " + lista[i]
+ ")");
                    }
                }
            }
            insertar(lista[i]);
        }
    }
}

```



Explicación:

Esta clase implementa una tabla hash diseñada para encontrar pares de elementos en una lista que sumen un valor dado. Utiliza una función de hash para determinar el índice de cada elemento en la tabla hash, gestionando colisiones mediante listas en cada bucket. El método insertar agrega elementos a la tabla verificando si ya existen y actualizando sus frecuencias, mientras que buscar determina la presencia de una clave buscada. La función encontrar_pares recorre la lista proporcionada para encontrar y devolver todos los pares de elementos que suman la cantidad deseada.

Implementar una Tabla Hash para Almacenar Valores Asociados a una Coordenada

```

package Data;

public class TablaHashCoordenadas {
    private int tamaño;
    private String[][] tabla;

    public TablaHashCoordenadas(int tamaño) {
        this.tamaño = tamaño;
    }
}

```

```

        this.tabla = new String[tamaño][];

    }

    private int hash(int x, int y) {
        return (x * 31 + y) % tamaño;
    }

    public void insertar(int x, int y, String valor) {
        int indice = hash(x, y);
        if (tabla[indice] == null) {
            tabla[indice] = new String[]{x + "," + y + ":" + valor};
        } else {
            String[] bucket = tabla[indice];
            String[] newBucket = new String[bucket.length + 1];
            System.arraycopy(bucket, 0, newBucket, 0, bucket.length);
            newBucket[bucket.length] = x + "," + y + ":" + valor;
            tabla[indice] = newBucket;
        }
    }

    public String buscar(int x, int y) {
        int indice = hash(x, y);
        if (tabla[indice] != null) {
            for (String elemento : tabla[indice]) {
                if (elemento.startsWith(x + "," + y + ":")) {
                    return elemento.substring(elemento.indexOf(":") + 1);
                }
            }
        }
        return null;
    }

    public String eliminar(int x, int y) {
        int indice = hash(x, y);
        if (tabla[indice] != null) {
            for (int i = 0; i < tabla[indice].length; i++) {
                if (tabla[indice][i].startsWith(x + "," + y + ":")) {
                    String valor =
tabla[indice][i].substring(tabla[indice][i].indexOf(":") + 1);
                    String[] newBucket = new String[tabla[indice].length - 1];
                    System.arraycopy(tabla[indice], 0, newBucket, 0, i);
                    System.arraycopy(tabla[indice], i + 1, newBucket, i,
tabla[indice].length - i - 1);
                    tabla[indice] = newBucket;
                    return valor;
                }
            }
        }
        return null;
    }
}

```

Explicacion:

Esta clase gestiona valores asociados a coordenadas (x, y) utilizando una tabla hash con manejo de colisiones mediante encadenamiento. La función de hash calcula un índice basado en las coordenadas de entrada, y insertar añade nuevos valores o actualiza los existentes en el bucket correspondiente. buscar localiza y devuelve el valor asociado a una coordenada específica, mientras que eliminar elimina el par clave-valor de la tabla hash si está presente. Esta implementación garantiza acceso eficiente a datos mediante el uso de estructuras de datos simples y métodos de búsqueda directa.

```

1 package Data;
2
3 public class TablaHashCoordinadas {
4     private int tamaño;
5     private String[][] tabla;
6
7     public TablaHashCoordinadas(int tamaño) {
8         this.tamaño = tamaño;
9         this.tabla = new String[tamaño][];
10    }
11
12    private int hash(int x, int y) {
13        return (x * 31 + y) % tamaño;
14    }
15
16    public void insertar(int x, int y, String valor) {
17        int indice = hash(x, y);
18        if (tabla[indice] == null) {
19            tabla[indice] = new String[]{x + ":" + y + ":" + valor};
20        } else {
21            String[] bucket = tabla[indice];
22            String[] newBucket = new String[bucket.length + 1];
23            System.arraycopy(bucket, 0, newBucket, 0, bucket.length);
24            newBucket[bucket.length] = x + ":" + y + ":" + valor;
25            tabla[indice] = newBucket;
26        }
27    }
28
29    public String buscar(int x, int y) {
30        int indice = hash(x, y);
31        if (tabla[indice] != null) {
32            for (String elemento : tabla[indice]) {
33                if (elemento.startsWith(x + ":" + y + ":")) {
34                    return elemento.substring(elemento.indexOf(":") + 1);
35                }
36            }
37        }
38    }
39
40    public void eliminar(int x, int y) {
41        int indice = hash(x, y);
42        if (tabla[indice] != null) {
43            for (String elemento : tabla[indice]) {
44                if (elemento.startsWith(x + ":" + y + ":")) {
45                    tabla[indice].remove(elemento);
46                }
47            }
48        }
49    }
50
51    public String[] getTabla() {
52        return tabla;
53    }
54
55    public void setTabla(String[][] tabla) {
56        this.tabla = tabla;
57    }
58
59    public int getTamaño() {
60        return tamaño;
61    }
62
63    public void setTamaño(int tamaño) {
64        this.tamaño = tamaño;
65    }
66}

```

Cuestionario

1. ¿Cuáles son las ventajas y desventajas de utilizar tablas hash en comparación con otras estructuras de datos? Explique.

Ventajas:

- Búsqueda Eficiente: La búsqueda en una tabla hash tiene tiempo promedio de acceso constante $O(1)$, lo cual es muy eficiente para operaciones de inserción, búsqueda y eliminación.
- Implementación Simple: Conceptualmente, las tablas hash son relativamente simples de implementar y entender, lo que facilita su uso en una variedad de aplicaciones.
- Adaptabilidad a Datos No Estructurados: Son ideales cuando la estructura de los datos no está claramente definida de antemano, como en el caso de claves arbitrarias o cadenas de texto.

Desventajas:

- Colisiones: Si dos claves distintas tienen el mismo valor hash, se produce una colisión. Manejar colisiones puede requerir técnicas adicionales como encadenamiento (usar listas enlazadas) o rehashing, lo que puede afectar el rendimiento.
- Espacio Adicional: En algunos casos, puede requerir más espacio que otras estructuras de datos debido a la necesidad de mantener una tabla de hash y estructuras adicionales para manejar colisiones.
- No Ordenación: No están diseñadas para mantener un orden específico de los elementos, por lo que no son adecuadas para operaciones que requieren iteración ordenada de los datos.

2. ¿Cuál es la diferencia entre una tabla hash estática y una tabla hash dinámica? Explique.

Tabla Hash Estática:

- Tiene un tamaño fijo que se determina al crear la tabla y no cambia durante su vida útil.
- Puede enfrentar problemas de capacidad si se llena, lo que puede requerir rehashing manual para aumentar el tamaño y redistribuir los elementos.

Tabla Hash Dinámica:

- Tiene un tamaño variable que puede ajustarse automáticamente según la cantidad de elementos y/o un factor de carga definido.
- Puede manejar automáticamente la redistribución y el manejo de colisiones mediante técnicas como el rehashing automático y el uso de estructuras de datos adicionales como listas enlazadas para el encadenamiento.

3. Haga una comparación entre la estructura de datos Tablas Hash y Árboles B, considerando los siguientes aspectos de cada una ellas que se muestran a continuación en la tabla siguiente:

Aspecto	Tablas Hash	Árboles B
Búsqueda	Tiempo promedio O(1)	Tiempo promedio O(log n)
Inserción y Eliminación	Tiempo promedio O(1)	Tiempo promedio O(log n)
Ordenación	No mantienen orden	Mantienen orden (en orden ascendente)
Espacio en Memoria	Puede requerir más espacio debido a estructuras extra	Requieren menos espacio en general que las tablas hash
Flexibilidad de Tamaño	Estático o dinámico	Dinámico (se ajustan automáticamente con la inserción)
Manejo de Colisiones	Necesita técnicas como encadenamiento o rehashing	No se aplican, ya que no hay colisiones
Aplicaciones Comunes	Bases de datos, tablas de dispersión	Bases de datos, sistemas de archivos, estructuras de índice

Referencias

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," 3rd ed., MIT Press, 2009.
- D. E. Knuth, "The Art of Computer Programming, Volume 3: Sorting and Searching," 2nd ed., Addison-Wesley, 1998.
- J. L. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," Proc. 8th ACM-SIAM Symposium on Discrete Algorithms, pp. 360-369, 1997.
- T. H. Cormen, "Hashing, Universal Hashing, and Perfect Hashing," MIT Course Lecture Notes, 2011.
- N. Ferguson, B. Schneier, and T. Kohno, "Cryptography Engineering: Design Principles and Practical Applications," Wiley, 2010.
- A. Joux, "Algorithmic Cryptanalysis," CRC Press, 2009.