	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I. Edgar Tista García

Asignatura: Estructura de datos y algoritmos II -1317

Grupo: 10

No. de práctica(s): 5

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de lista o brigada: 26

Semestre: 2023-1

Fecha de entrega: 21 de octubre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 5.

Algoritmos de búsqueda. Parte 2

OBJETIVOS

- **Objetivo general:** El estudiante conocerá e identificará las características necesarias para realizar búsquedas por **transformación de llaves**.
- **Objetivo de la clase:** El alumno conocerá la implementación de la transformación de llaves en el lenguaje orientado a objetos.

DESARROLLO

Instrucciones

Ejercicio 0. Menú para la práctica

a) Con ayuda de NetBeans, crea un nuevo proyecto llamado Practica5[ApPaternoNombre].
b) Crea un menú en el cual puedas seleccionar entre las opciones siguientes
    1) Manejo de Tablas Hash en Java
    2) Función Hash por módulo
    3) Encadenamiento

Ejercicio 1. Tablas Hash en Java

El manejo de Tablas Hash en el lenguaje de programación Java se realiza a través de la interfaz Map<E>. Dicha interfaz cuenta con varias implementaciones. Las más usadas son HashMap y TreeMap, la diferencia fundamental entre ambas es que la primera permite elementos nulos, conserva un cierto orden en su estructura y no es “sincronizada”; en tanto que la segunda no permite elementos nulos, los elementos no tienen un orden específico y maneja la sincronización.

Al igual que la práctica anterior, deberás crear una Tabla Hash en Java (puede ser HashMap oTreeMap). Para la tabla hash deberás crear parejas donde el valor será un *String* que representa el nombre (completo) de un alumno y la clave de la tabla hash será un número entero que representa el número de cuenta.

Agrega algunos elementos a la tabla hash. Y posteriormente, con la tabla creada prueba los siguientes métodos:

- contains
- containsKey
- containsValue
- equals
- get
- put
- remove
- size

Si lo deseas puedes agregar otros métodos de Tablas Hash y explicar su funcionamiento.

Ejercicio 2. Simulación de Función Hash por plegamiento y módulo.

a) Agrega una clase al proyecto llamada HashModulo, en ella, crea una nueva lista (puede ser LinkedList o ArrayList) de enteros. Para efectos de la práctica se manejará una lista de tamaño fijo de 15 elementos (Para fijar este tamaño, al inicio se deben inicializar todos sus elementos con null).

   Elabora un submenú para que el usuario pueda seleccionar entre las siguientes opciones

   1) Agregar elementos
   2) Imprimir lista
   3) Buscar elementos

b) Crea un método donde se utilizará la función hash por plegamiento, tal y como se realiza en el ejercicio de clase (pliegues de tamaño 4).

   - ✓ Los elementos a ingresar son claves de 9 dígitos de longitud.
   - ✓ Se deberá hacer la suma del plegamiento y posteriormente la operación de módulo % 15 para determinar la posición de la lista donde se va a almacenar.
   - ✓ En caso de haber colisiones, se resolverán por el método de PRUEBA LINEAL.

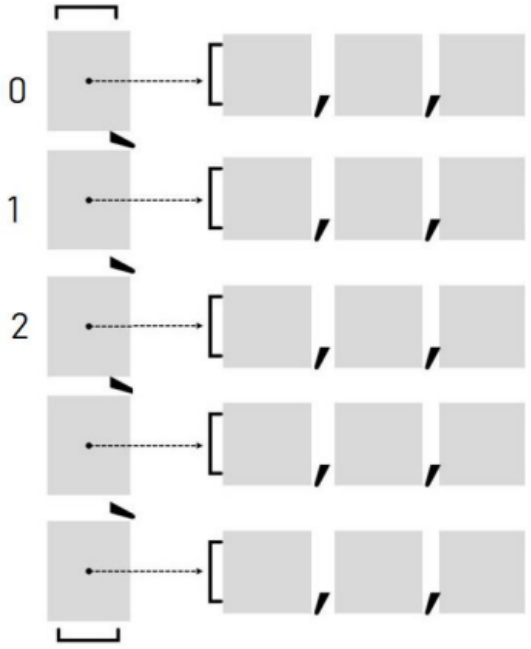
c) Al agregar o buscar elementos el programa deberá informar al usuario el resultado de la función Hash aplicada indicando la posición donde se almaceno/encontró el elemento.

Ejercicio 3. Encadenamiento

Agrega una nueva clase al proyecto llamada *Encadenamiento*, en ella deberás realizar los siguiente:

a) Deberás crear una lista de 15 listas de enteros.
b) Implementa una función aleatoria que determinara la posición de los elementos ingresados por el usuario (La función aleatoria solamente entregará un número entre 0 y 14).

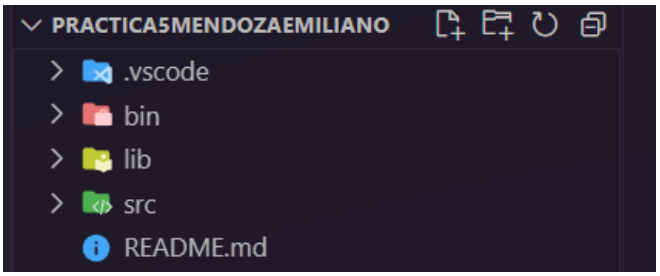
El usuario podrá elegir, agregar elemento o salir de la sección. Cada vez que el usuario ingrese un número, la función aleatoria indicará la posición donde se ubicará el número y mostrará el estado global de la lista.



Descripción de actividades realizadas

1) Se creó el proyecto *Practica5MendozaEmiliano* usando el editor de texto Visual Studio Code.

   Creación de proyecto java:



2) Se creó la clase *Main* dentro de la carpeta *src* para implementar en él el menú y llevar el flujo de ejecución del programa

   Menú principal:

```
java.util.Scanner stdin = new java.util.Scanner(System.in);
        System.out.println("***** Menú Práctica 5 *****");
        System.out.println("Selecciona una opción");
        System.out.println("1) Manejo de Tablas Hash en Java");
        System.out.println("2) Función Hash por módulo");
        System.out.println("3) Encadenamiento");
        System.out.print("Opción: ");
        int opc = stdin.nextInt();
```

   En estas líneas se imprime el menú y se selecciona la opción ingresada por el usuario.

3) Se implementó una estructura *switch-case* para manejar el flujo de ejecución desde el menú principal

   Case 1:

   En esta opción del menú se pretende interactuar con Tablas Hash, creando alguna instancia de *HashMap* o *TreeMap* y probando sus principales métodos. Para este caso en particular, se decidió usar *HashMap*.

   Dado que las claves serán números de cuenta los y los valores serán nombres de los alumnos, se declaró un nuevo *HashMap* llamado *myHashMap* de la siguiente manera:

```
HashMap< Integer, String > myHashMap = new HashMap<>();
```

Se probó el método ***put*** (`V put(K key, V value)`) para agregar algunos elementos a *myHashMap*.

Algunos de los elementos agregados son los siguientes:

```
myHashMap.put(912419031, "Armando Archundia");
myHashMap.put(872340194, "Erling Haaland");
myHashMap.put(948103314, "Teresa Fidalgo");
myHashMap.put(481048952, "Selene Delgado");
```

Se creó otro *HashMap* llamado *myHashMap2* para poder probar el método ***equals*** (`boolean equals(Object o)`). Para ello, se usó el constructor `HashMap(Map<? extends K,? extends V> m)`.

```
HashMap<Integer, String> myHashMap2 = new HashMap<>(myHashMap);
```

Este nuevo *HashMap* tiene las mismas claves y valores que *myHashMap*. Por lo tanto, el método ***equals*** debería devolver *true*.

```
myHashMap.equals(myHashMap2);
```

Se agregó otro elemento a *myHashMap* para así volver a probar el método ***equals***. Se espera que al modificar *myHashMap* la comparación ahora devuelva *false*.

```
myHashMap.put(824095247, "Hugo Boss");
myHashMap.equals(myHashMap2);
```

Se probó el método ***containsKey*** (`boolean containsKey(Object key)`) y el método ***containsValue*** (`boolean containsValue(Object value)`) para buscar una clave y un valor respectivamente.

```
myHashMap.containsKey(317579439);
myHashMap.containsValue("Erling Haaland");
```

Se probó el método ***get*** (`V get(Object key)`) usando claves para recuperar valores.

```
myHashMap.get(948103314);
myHashMap.get(317579439);
```

Se probó el método ***remove*** (`V remove(Object key)`) para eliminar elementos de *myHashMap*.

```
myHashMap.remove(912419031);
```

Sin embargo, la interfaz *Map* tiene otra versión del método ***remove*** (`boolean remove (Object key, Object value)`) que de igual manera se comprobó su funcionamiento.

```
myHashMap.remove(872340194, "Erling Haaland");
myHashMap.remove(817409810, "Hernan Cortes");
```

Se probó el método ***size*** (`int size()`) para conocer el tamaño de las estructuras creadas hasta este momento.

```
myHashMap.size();
myHashMap2.size();
```

Adicionalmente, se realizaron pruebas con el método ***replace***, el cual es un método sobrecargado de la interfaz *Map. P*uede utilizarse de estas dos formas: `V replace (K key, V value)` y `boolean replace(K key, V oldValue, V newValue)`.

```
myHashMap.replace(948103314, "El mencho");
myHashMap.replace(824095247, "Hugo Boss", "Aristoteles");
```

Por último, se probó el método ***isEmpty*** (`boolean isEmpty()`) para saber si *myHashMap* está vacía.

```
myHashMap.isEmpty();
```

4) Se creó la clase *HashModulo*. Esta clase tiene 1 atributo, 1 constructor y 3 métodos.

Atributo de la clase:

```
private LinkedList<Integer> lista = new LinkedList<>();
```

El único atributo de la clase es una *LinkedList* de valores enteros.

Constructor de la clase:

```
public HashModulo() {
        for (int i = 0; i < 15; i++) {
            lista.add(null);
        }
}
```

Este constructor “fija” el tamaño de la lista a 15 elementos, inicializando todos con *null*.

Métodos de la clase:

- El método *imprimirLista()* permite visualizar en forma de tabla todos las claves de la lista (que corresponden a los índices de la lista) y sus respectivos valores.

```
public void imprimirLista() {
        System.out.println("\nLa lista es:");
        System.out.println("Clave      Valor");
        for (int i = 0; i < 15; i++) {
            System.out.println(i + "           " + lista.get(i));
        }
}
```

- El método *modulo()* realiza la operación de la función hash módulo a un valor dado como parámetro y regresa el resultado de dicha operación. De acuerdo con la definición de función hash módulo:

$$h(x) = |x| \bmod M$$

Para este ejercicio se utiliza un arreglo de 15 elementos como contenedor. Por lo tanto, se utiliza la función hash módulo con $M = 15$. Además, con la condición del operador ternario se valida el resultado en caso de recibir algún número negativo.

```
public int modulo(int elemento) {
        return elemento < 0 ? (elemento * -1) % 15 : elemento % 15;
}
```

- Dado un elemento numérico como parámetro, el método *plegamiento()* realiza las operaciones correspondientes de la función hash por plegamiento de 4 dígitos, tomando el dígito menos significativo como resultado, que es devuelto por la función.

  Nota: Para esta función se asume que el usuario ingresará valores válidos (claves numéricas de 9 dígitos de longitud).

  En las primeras líneas del método se crea una copia de tipo *String* con el valor numérico dado. Se crea también un arreglo que contendrá 3 *Strings* (para los dígitos de la posición 0 a 3, de 4 al 7 y 8). Después se crea también una variable tipo *String* que servirá como variable temporal para agregar los números en forma de cadena al arreglo.

```
public int plegamiento(int elemento) {
        String elementS = String.valueOf(elemento);
        String sub[] = new String[3];
        String temp = "";
```

Concatena los dígitos de la posición 0 a la 3 en la variable *temp*, luego la agrega al arreglo en la primera posición.

```
        for (int i = 0; i < 4; i++)
            temp += elementS.charAt(i);
        sub[0] = temp;
```

"Vacía" la variable *temp*. Luego, concatena los dígitos de la posición 4 a la 7 en la variable *temp*, y la agrega al arreglo en la segunda posición.

```
        temp = "";
        for (int i = 4; i < 8; i++)
            temp += elementS.charAt(i);
        sub[1] = temp;
```

"Vacía" la variable *temp* una vez más. Luego, agrega el dígito de la posición 8 al arreglo en la tercera posición.

```
        temp = "";
        temp += elementS.charAt(8);
        sub[2] = temp;
```

Una vez “separado” el número original, se hace la suma de los plegamientos. Después de la suma, se vuelve a crear una variable tipo *String* con el resultado.

```
        int sum = 0;
        for (String string : sub) {
            int num = Integer.parseInt(string);
            sum += num;
        }
        String sumS = String.valueOf(sum);
```

Por último, se debe regresar un valor de tipo entero, pero como nuestra representación numérica es de tipo *String*, se usa la función *charAt* para recuperar el dígito en la última posición (tamaño de la cadena -1). Adicionalmente, se usa la función *getNumericValue* para “convertir” el carácter recuperado con *charAt* a tipo entero.

```
        return Character.getNumericValue(sumS.charAt(sumS.length() - 1));
}
```

- El método *agregarElemento()* está diseñado para ser invocado desde el menú principal, recibiendo como parámetro un valor numérico dado para hacer las operaciones hash a dicho parámetro (primero por plegamiento y al resultado se le aplica el módulo).

```
public void agregarElemento(int elemento) {
        int indice = plegamiento(elemento);
        indice = modulo(indice);
```

Luego, se debe verificar que el resultado obtenido (que es el índice en el arreglo donde se almacenará el elemento) no esté duplicado. En caso de que esto ocurra (una colisión), se resuelve de la siguiente manera: sabemos de antemano que aquellos índices donde aún no se ha almacenado algún número contienen *null*, por lo tanto, si el índice donde se quiere guardar un nuevo elemento contiene *null* previamente, el número se puede guardar sin problema. Caso contrario, si en el índice donde se quiere guardar el número ya contiene otro valor (es decir, es diferente de *null*) se debe incrementar el valor del índice en 1 para verificar la siguiente posición del arreglo. Esto se repetirá hasta que se encuentre algún índice que esté “vacío”. Sin embargo, es posible deducir dos cosas.

Primero: al llegar al final del arreglo, se debe regresar a la primera posición. Segundo: en caso de que el arreglo esté lleno, se debe detener el ciclo y evitar un bucle infinito. Todo lo anterior se condensa en el siguiente ciclo:

```
        int flag = 0;
        while (lista.get(indice) != null) {
            indice++;
            flag++;
            if (indice == 15)
                indice = 0;
            if (flag == 15)
                break;
        }
```

Donde *flag* es una bandera que permite saber cuántas veces se ha recorrido el índice para encontrar un lugar en el arreglo. Al llegar a 15, se rompe el ciclo dado que ya se habría recorrido todo el arreglo y no se encontró un índice disponible. En caso de que *flag* no haya llegado a 15, significa que hay un lugar disponible para guardar el arreglo.

```
        if (flag == 15) {
            System.out.println("ERROR: No se pudo agregar el elemento. Contenedor
lleno.");
        } else {
            lista.set(indice, elemento);
            System.out.println(elemento + " agregado exitosamente en " + indice);
        }
    }
```

- El último método de la clase es *bucarElemento()*, el cual regresa el índice de la lista donde se encuentra el elemento dado como parámetro. Si el elemento no está en la lista devuelve -1.

```
public int buscarElemento(int elemento) {
        return lista.contains(elemento) ? lista.indexOf(elemento) : -1;
    }
```

5) Se codificó la segunda opción del menú principal, implementando los métodos que se crearon en la clase *HashModulo*.

Case 2:

Se implementó un submenú con las siguientes opciones:

Submenú Función Hash por módulo:

```
char opcion2;
HashModulo hm = new HashModulo();
do {
    System.out.println("\nSelecciona una opción");
    System.out.println("a) Agregar elemento");
    System.out.println("b) Imprimir lista");
    System.out.println("c) Buscar elemento");
    System.out.println("d) Salir");
    System.out.print("Opción: ");
    opcion2 = stdin.next().charAt(0);
    int num;
    switch (opcion2) {
```

Observación: Es importante notar que el menú se ejecutará mientras el usuario decida seguir haciendo operaciones (*do – while*), por lo tanto, se debe instanciar la clase *HashModulo* antes de entrar al ciclo, de lo contrario (instanciando adentro del ciclo) se crearía una nueva instancia para cada ejecución del ciclo.

Las opciones del submenú se describen a continuación:

Case a:

Pide y lee en el teclado un elemento dado por el usuario, luego, se agrega el elemento a la lista.

```
System.out.print("\nIngrese elemento: ");
num = stdin.nextInt();
hm.agregarElemento(num);
```

Case b:

Imprime el estado actual de la lista.

```
hm.imprimirLista();
```

Case c:

Pide y lee en el teclado un elemento dado por el usuario, luego, si el elemento está en la lista, indica el índice en el que se encuentra. En caso contrario, indica que el elemento no está en la lista.

```
System.out.print("\nIngrese elemento: ");
num = stdin.nextInt();
if (hm.buscarElemento(num) == -1)
    System.out.println(num + " no se encuentra en la lista");
else
    System.out.println(num + " esta en el indice " + hm.buscarElemento(num));
```

Case d:

Termina la ejecución del programa.

6) Se creó la clase *Encadenamiento*. Esta clase tiene 1 atributo, 1 constructor y 2 métodos.

Atributo de la clase:

```
private LinkedList<LinkedList<Integer>> listaDeListas = new LinkedList<>();
```

Es una lista que contiene a su vez listas de tipo entero.

Constructor de la clase:

```
public Encadenamiento() {
        for (int i = 0; i < 15; i++) {
            listaDeListas.add(new LinkedList<Integer>());
        }
}
```

Inicializa 15 listas vacías de enteros en la lista de listas.

Métodos de la clase:

- El método *agregarElemento()* pide un valor por teclado por al usuario, y elige de manera aleatoria (con apoyo del método *Random.nextInt()*) en qué lista se agregará el valor. El número aleatorio es un número entre 0 y 14, que es el índice que contiene la lista donde se almacenará el nuevo valor.

```
public void agregarElemento(int elemento) {
        Random rand = new Random();
        int indice = rand.nextInt(0,15);
        listaDeListas.get(indice).add(elemento);
        System.out.println(elemento + " se agrego a la lista " + indice);
        this.imprimirLista();
}
```

- El método *imprimirLista()* permite visualizar las sublistas de la lista de listas y los elementos que contienen.

```
private void imprimirLista() {
        System.out.println("\nIndice      Sublista");
        for (int i = 0; i < 15; i++) {
            System.out.println(i + "           " + listaDeListas.get(i));
        }
    }
```

7) Después, se agregaron las instrucciones para la tercera opción del menú principal.

Case 3:

En esta opción, se pretende interactuar con una lista de listas de enteros que guardará en localidades aleatorias los números ingresados por el usuario. Para ello, se creó otro pequeño submenú con las opciones del usuario.

Submenú de Encadenamiento:

```
char opcion3;
Encadenamiento e = new Encadenamiento();
do {
    System.out.println("\nSelecciona una opción");
    System.out.println("a) Agregar elemento");
    System.out.println("b) Salir");
    System.out.print("Opción: ");
    opcion3 = stdin.next().charAt(0);
    int num;
    switch (opcion3) {
```

Observación: Al igual que en el submenú de la segunda opción, es necesario instanciar la clase encadenamiento antes de entrar al ciclo *do-while*.

Case a: Al ingresar a esta opción, se solicita un número al usuario y se agrega usando el método *agregarElemento()* de la clase *Encadenamiento*.

```
System.out.print("\nIngrese elemento: ");
num = stdin.nextInt();
e.agregarElemento(num);
```

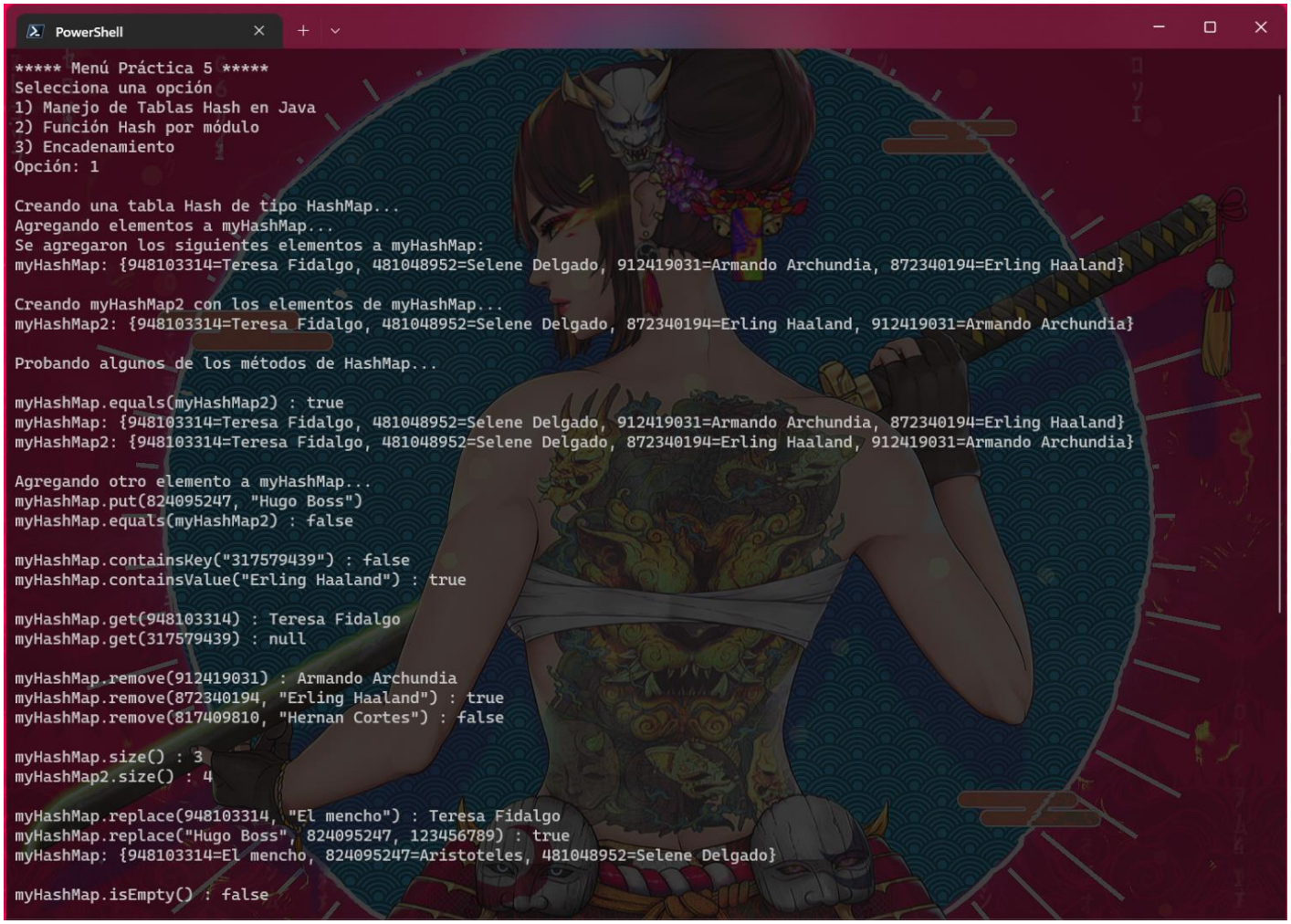
Case b: Termina la ejecución del programa.

8) Una vez concluida la codificación de las clases y el *main*, se procedió a hacer las pruebas y ejecución del programa.

RESULTADOS OBTENIDOS

Se obtuvo la siguiente salida al ejecutar el programa y elegir la opción 1:

Ejecución en terminal

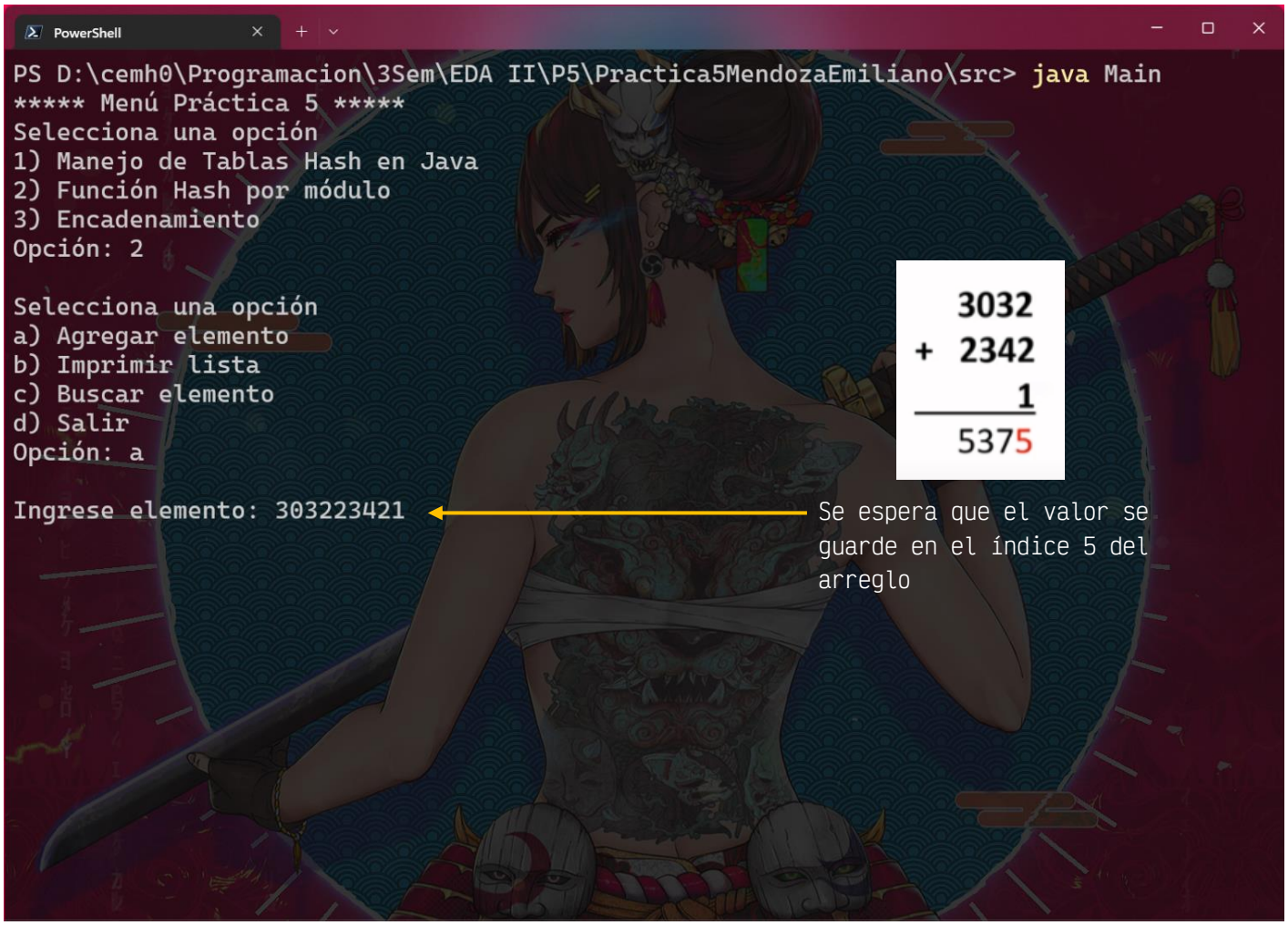


Observaciones: Para crear y hacer uso de Tablas Hash en Java, *HashMap* es una buena opción porque sus métodos implementados facilitan mucho su manipulación. Revisando un poco la API es posible saber para qué utilizarlos, de qué manera invocarlos, que parámetros requieren y que valores devuelven. En este ejemplo se utilizaron algunos métodos sobrecargados que, si bien, tienen el mismo nombre, tienen algunas diferencias entre sí (por ejemplo, el método *remove* y el método *replace)* como la cantidad de parámetros que reciben y el tipo de valor de retorno. Además, es relativamente sencillo buscar una clave o un valor en un *HashMap*.

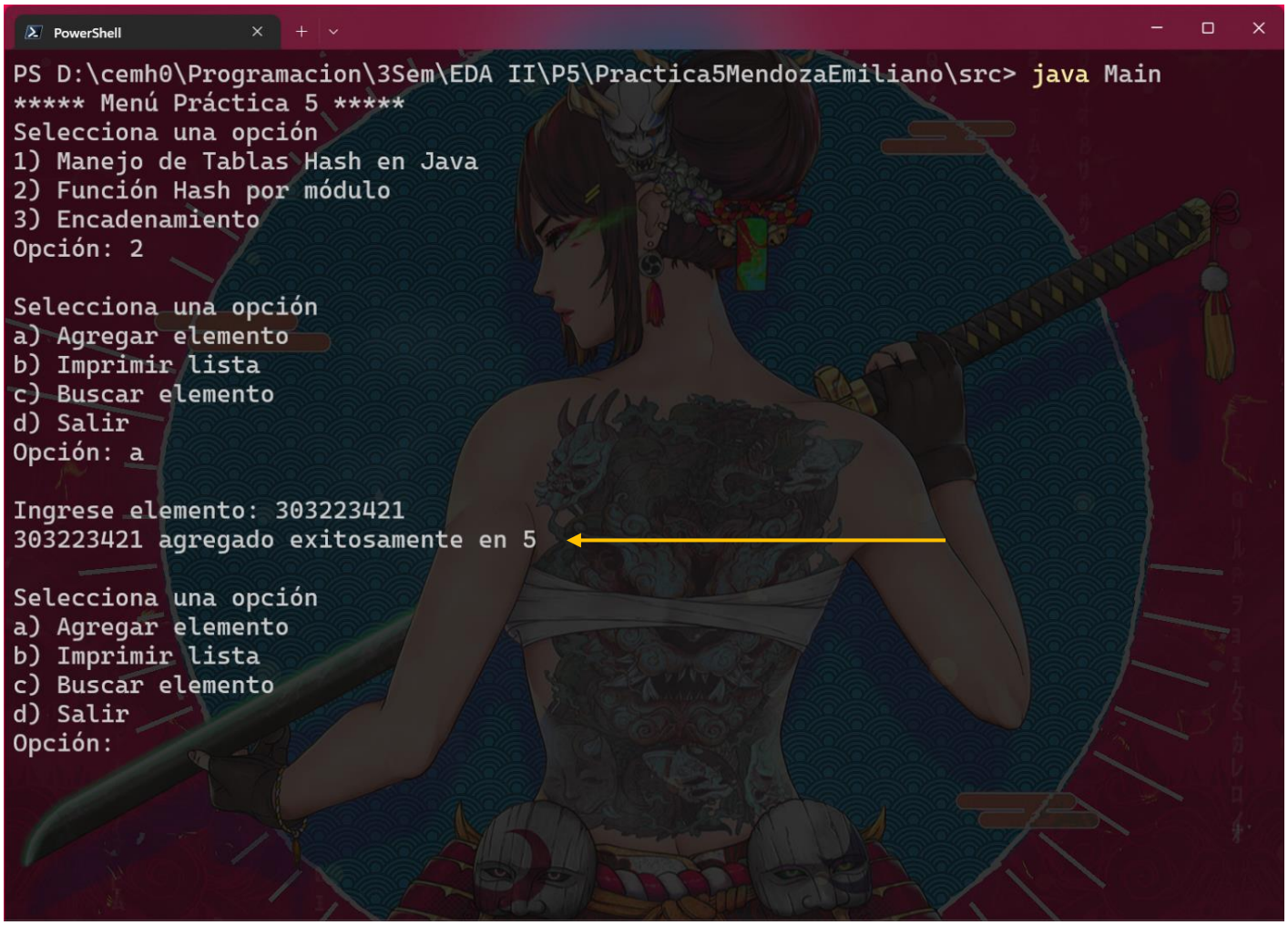
Otra característica de los *HashMap* que se puede observar en la prueba de los métodos es que los pares clave-valor no están ordenados de alguna forma en particular. Por ello, para que dos *HashMap* sean iguales basta con que contengan las mismas claves asociadas a los mismos valores, ya que el orden es irrelevante en estas estructuras.

Ahora, se ejecuta nuevamente y eligiendo la segunda opción del menú principal.

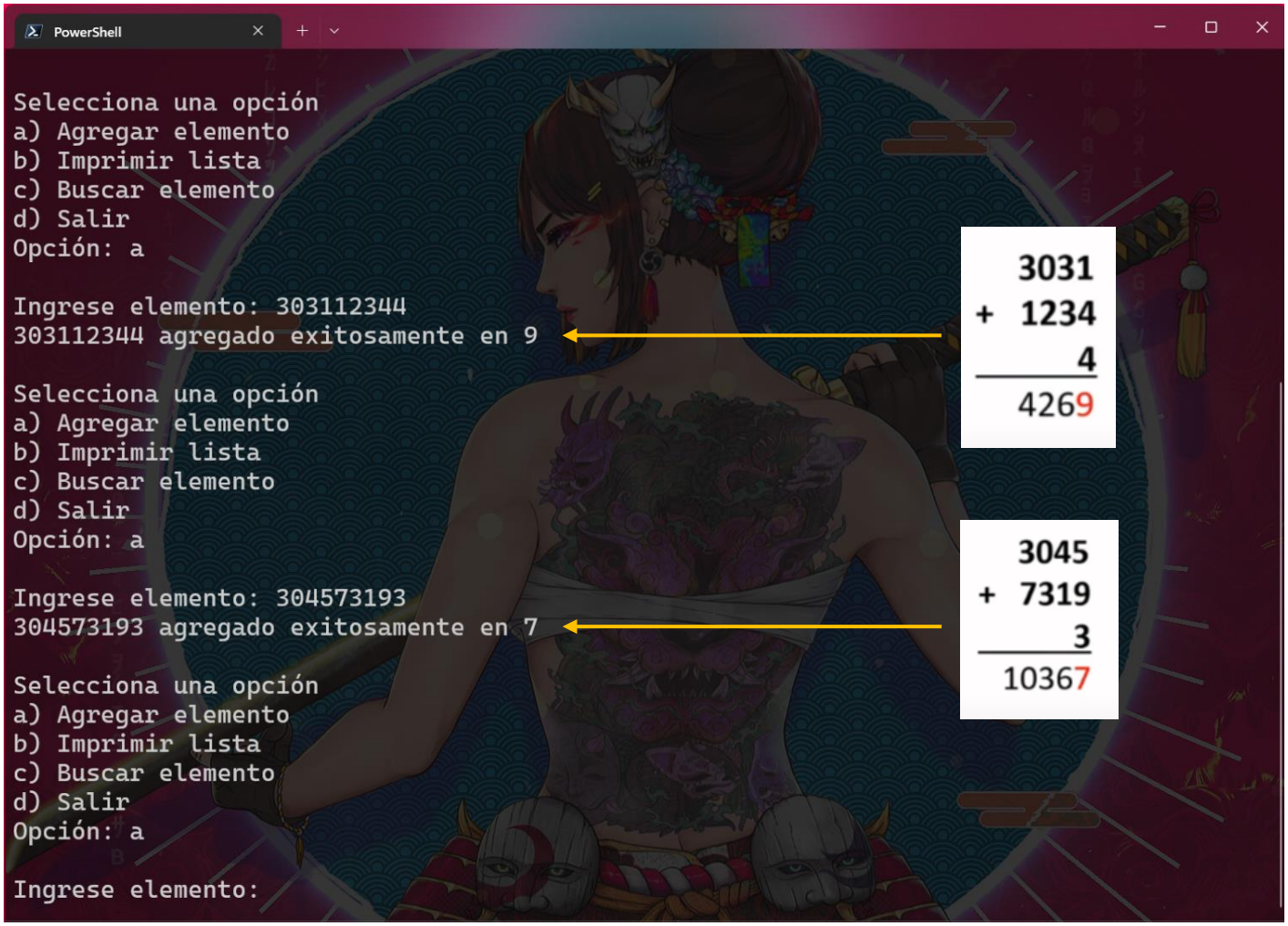
Ejecución en terminal

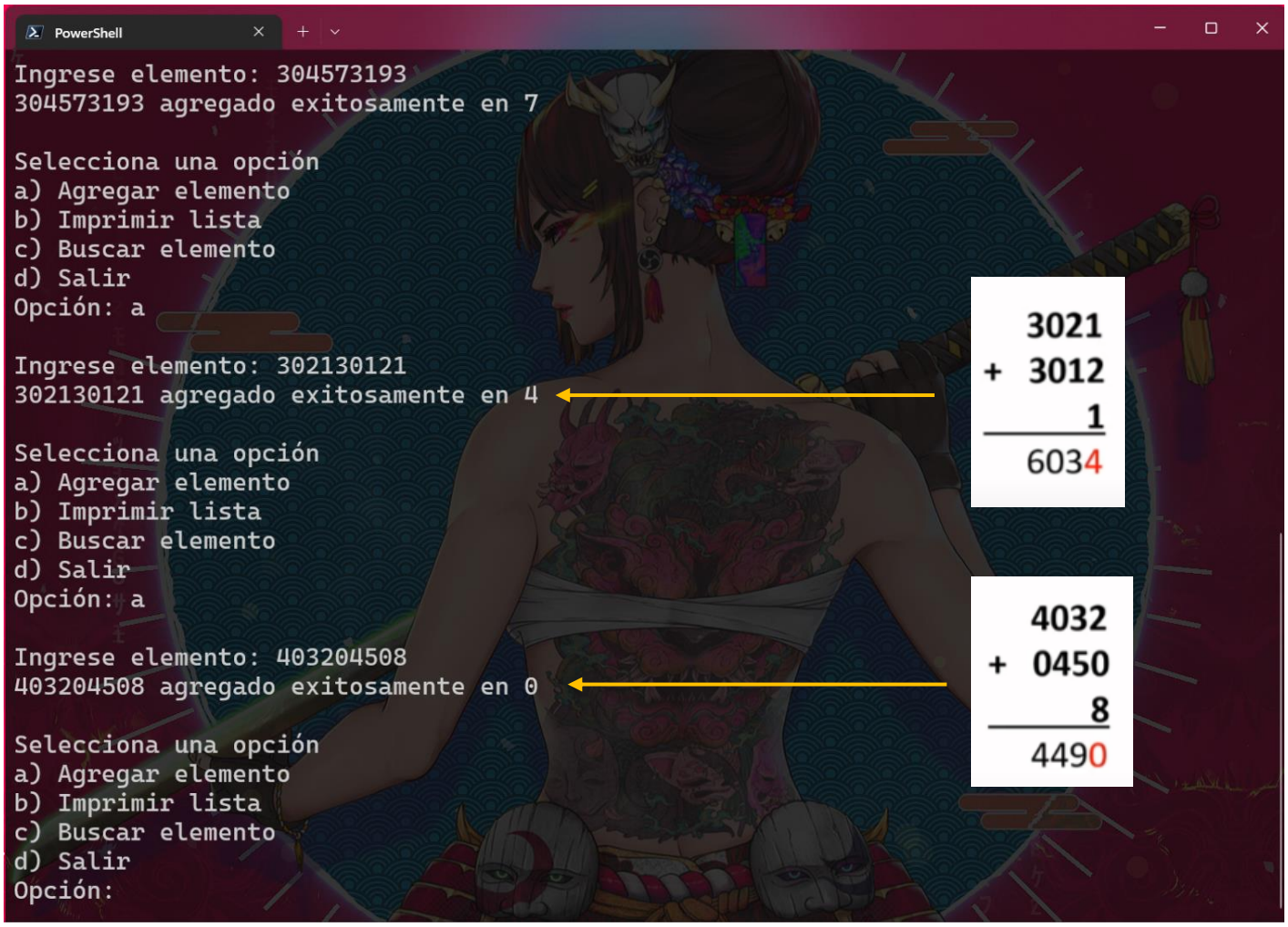


Observación: Dado que al hacer la función hash por plegamiento se toma el dígito menos significativo (0-9), el resultado de aplicar la función hash módulo 15 no alterará los valores que resulten de hacer el plegamiento.

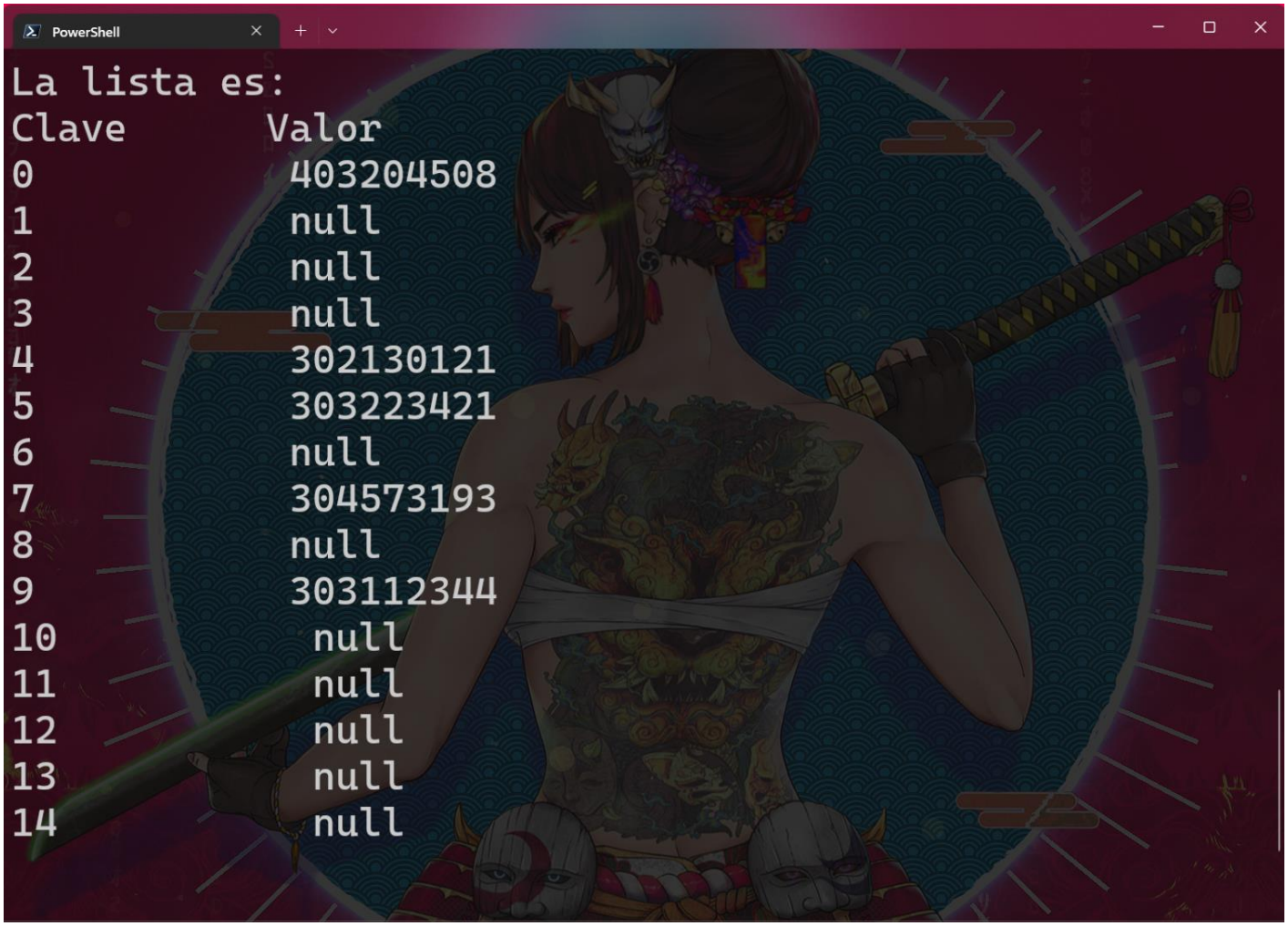

PowerShell
PS D:\cemh0\Programacion\3Sem\EDA II\P5\Practica5MendozaEmiliano\src> java Main
***** Menú Práctica 5 *****
Selecciona una opción
1) Manejo de Tablas Hash en Java
2) Función Hash por módulo
3) Encadenamiento
Opción: 2
Selecciona una opción
a) Agregar elemento
b) Imprimir lista
c) Buscar elemento
d) Salir
Opción: a
Ingrese elemento: 303223421
303223421 agregado exitosamente en 5
Selecciona una opción
a) Agregar elemento
b) Imprimir lista
c) Buscar elemento
d) Salir
Opción:

Guardando más valores en el arreglo:



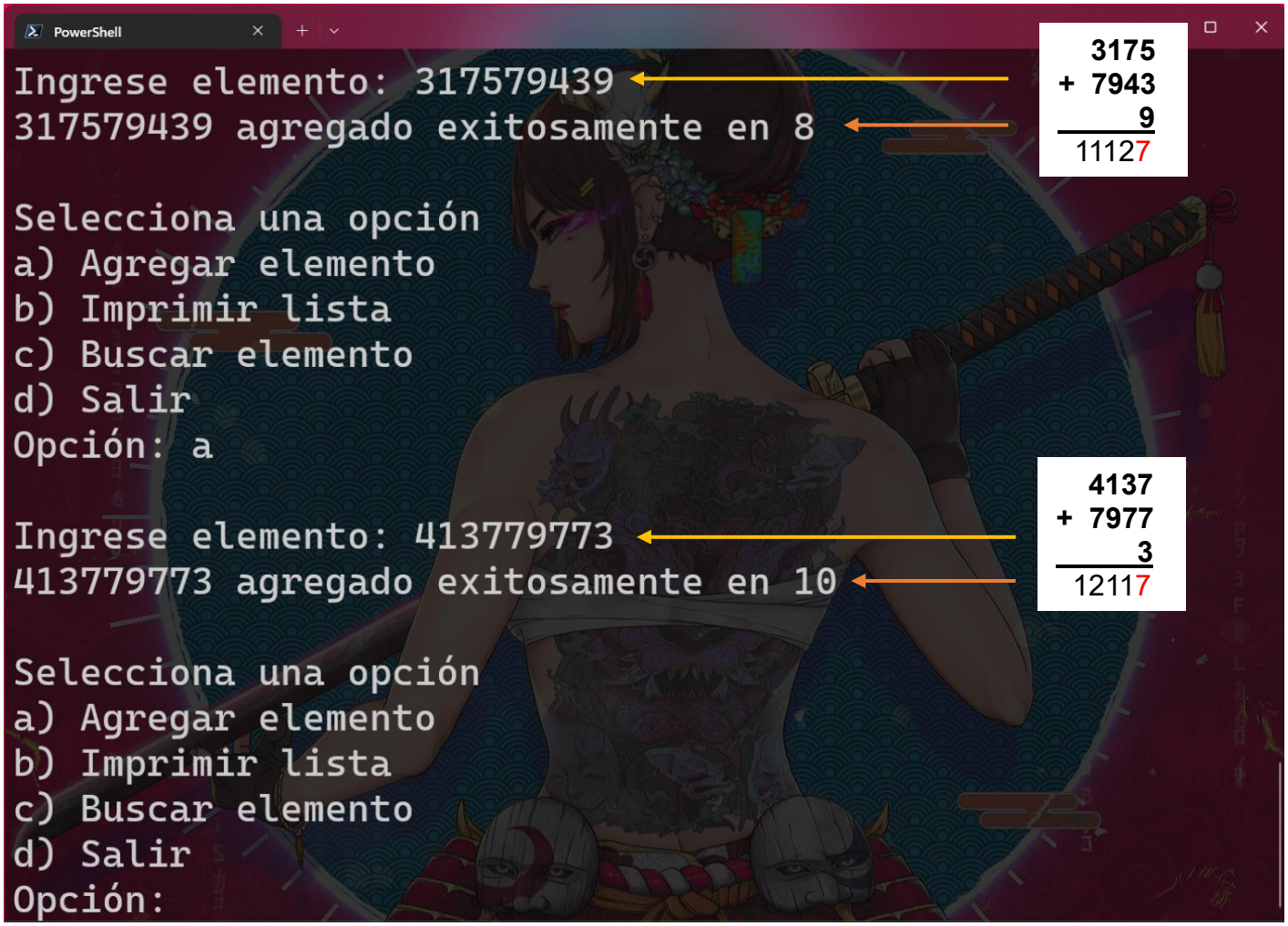

PowerShell
Ingrese elemento: 304573193
304573193 agregado exitosamente en 7
Selecciona una opción
a) Agregar elemento
b) Imprimir lista
c) Buscar elemento
d) Salir
Opción: a
Ingrese elemento: 302130121
302130121 agregado exitosamente en 4
Selecciona una opción
a) Agregar elemento
b) Imprimir lista
c) Buscar elemento
d) Salir
Opción: a
Ingrese elemento: 403204508
403204508 agregado exitosamente en 0
Selecciona una opción
a) Agregar elemento
b) Imprimir lista
c) Buscar elemento
d) Salir
Opción:
3021
+ 3012
1
6034
4032
+ 0450
8
4490

Imprimiendo el arreglo actual observamos las localidades que se han ocupado:



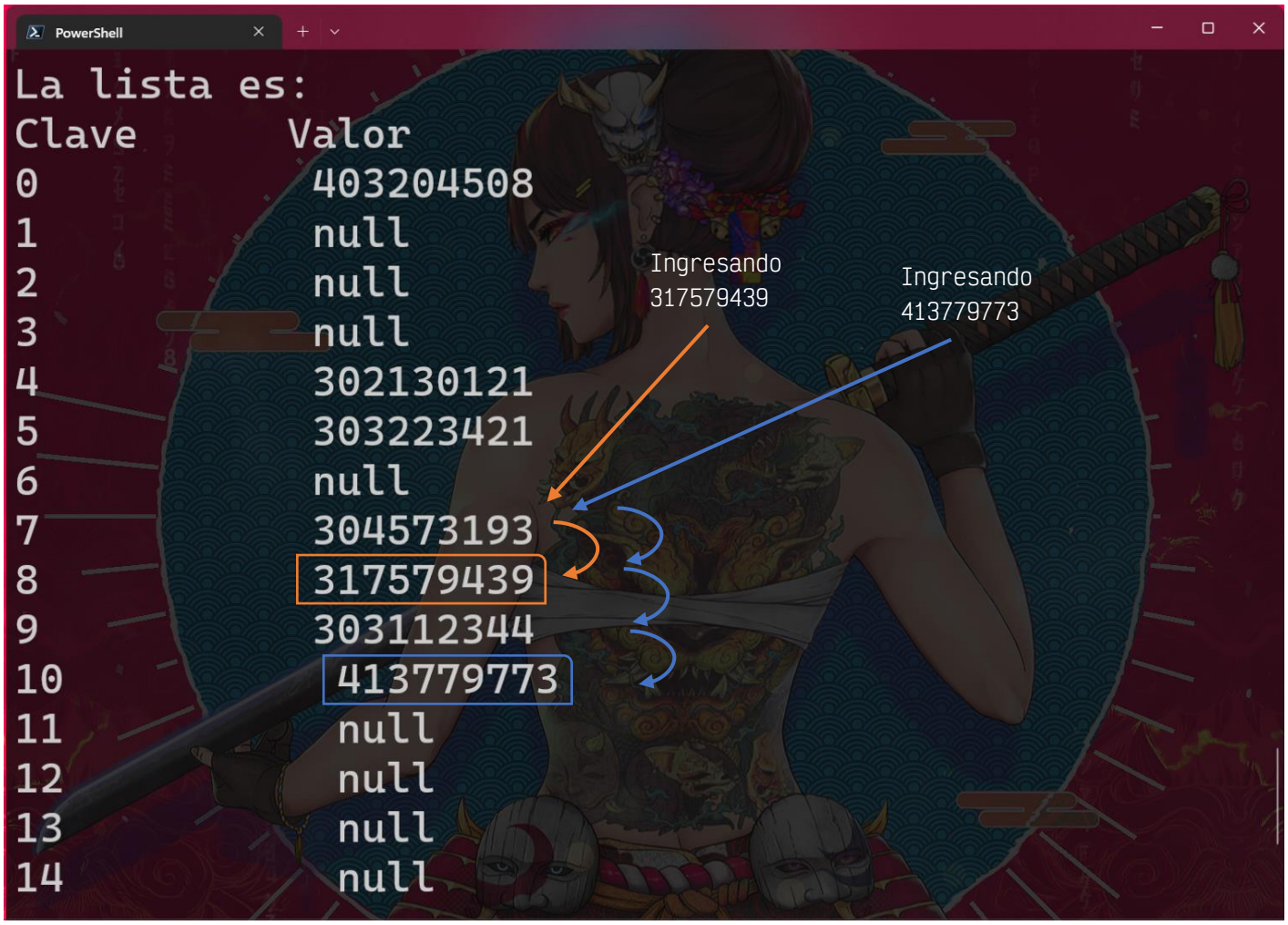
0	1	2	3	4	5	6	7	8	9
403204508				302130121	303223421		304573193		303112344

A continuación, se introducirán números de cuenta que generen claves repetidas:



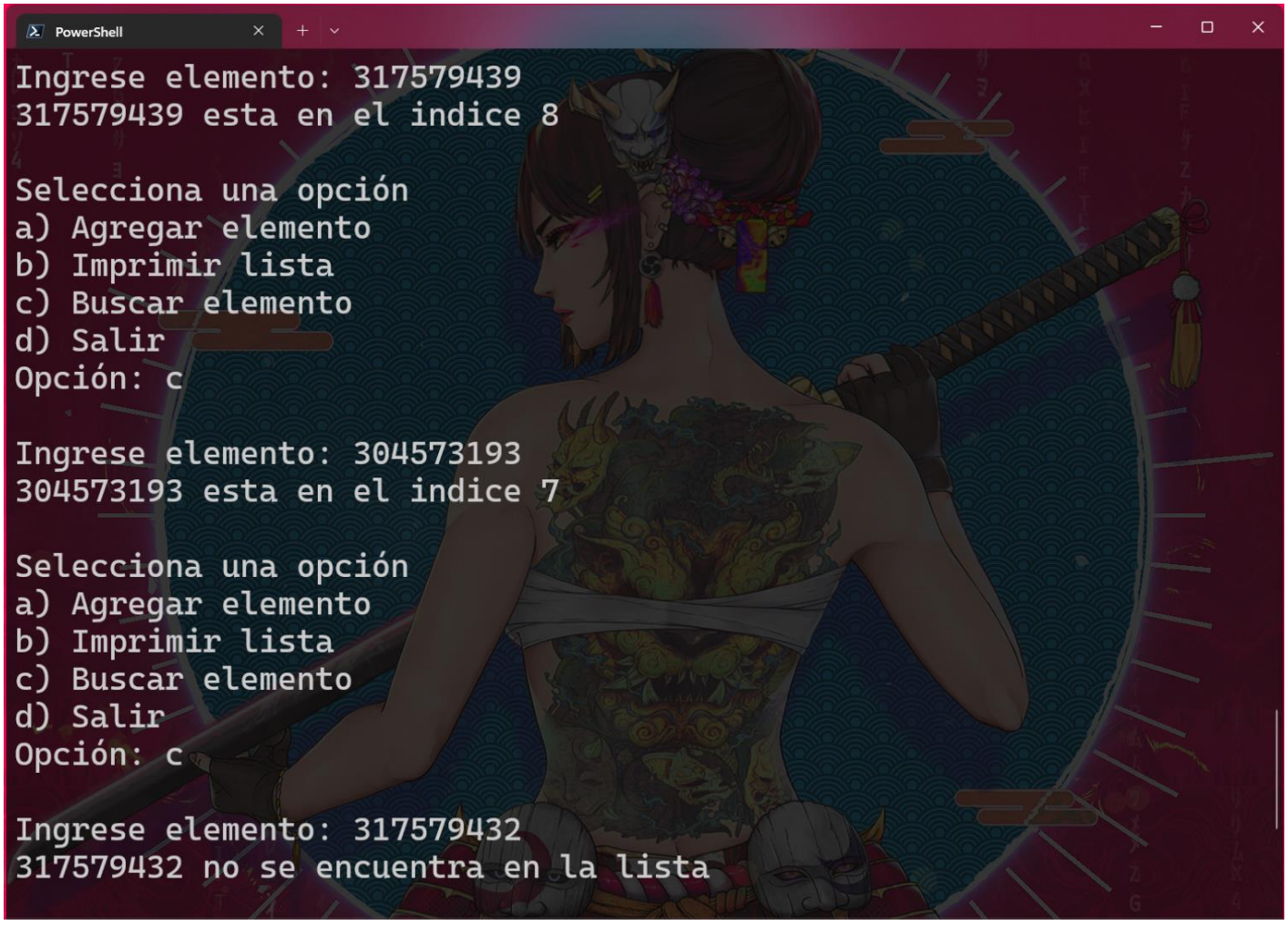
Observaciones: Con ambos números, se obtiene 7 como resultado al hacer la función hash, sin embargo, el índice 7 ya contiene un elemento, y es por esto por lo que se les asigna un nuevo índice con la prueba lineal (también se debe a que aún hay espacio en el contenedor para alojar estos números).

Al imprimir nuevamente la lista es más fácil observar cómo se incrementa el índice de 317579439 hasta encontrar un lugar en la posición 8 y cómo se incrementa el índice de 413779773 hasta encontrar un lugar en la posición 10:

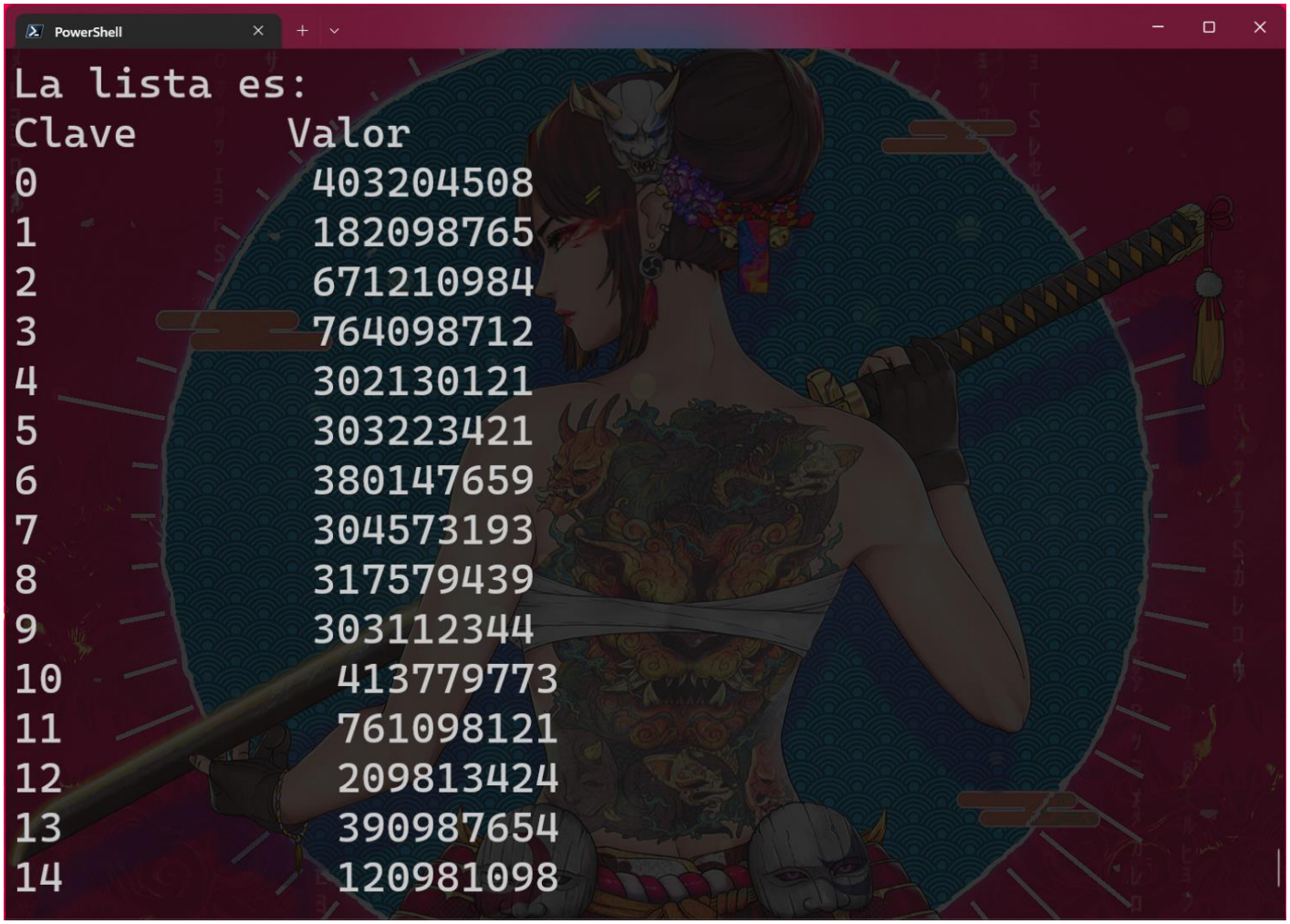


Observaciones: El método de prueba lineal condensa los valores en un “espacio” reducido. Es decir, no distribuye uniformemente los valores en el contenedor, provocando un “espacio vacío” relativamente grande entre 11 y 14 y un “espacio lleno” relativamente grande entre 7 y 10.

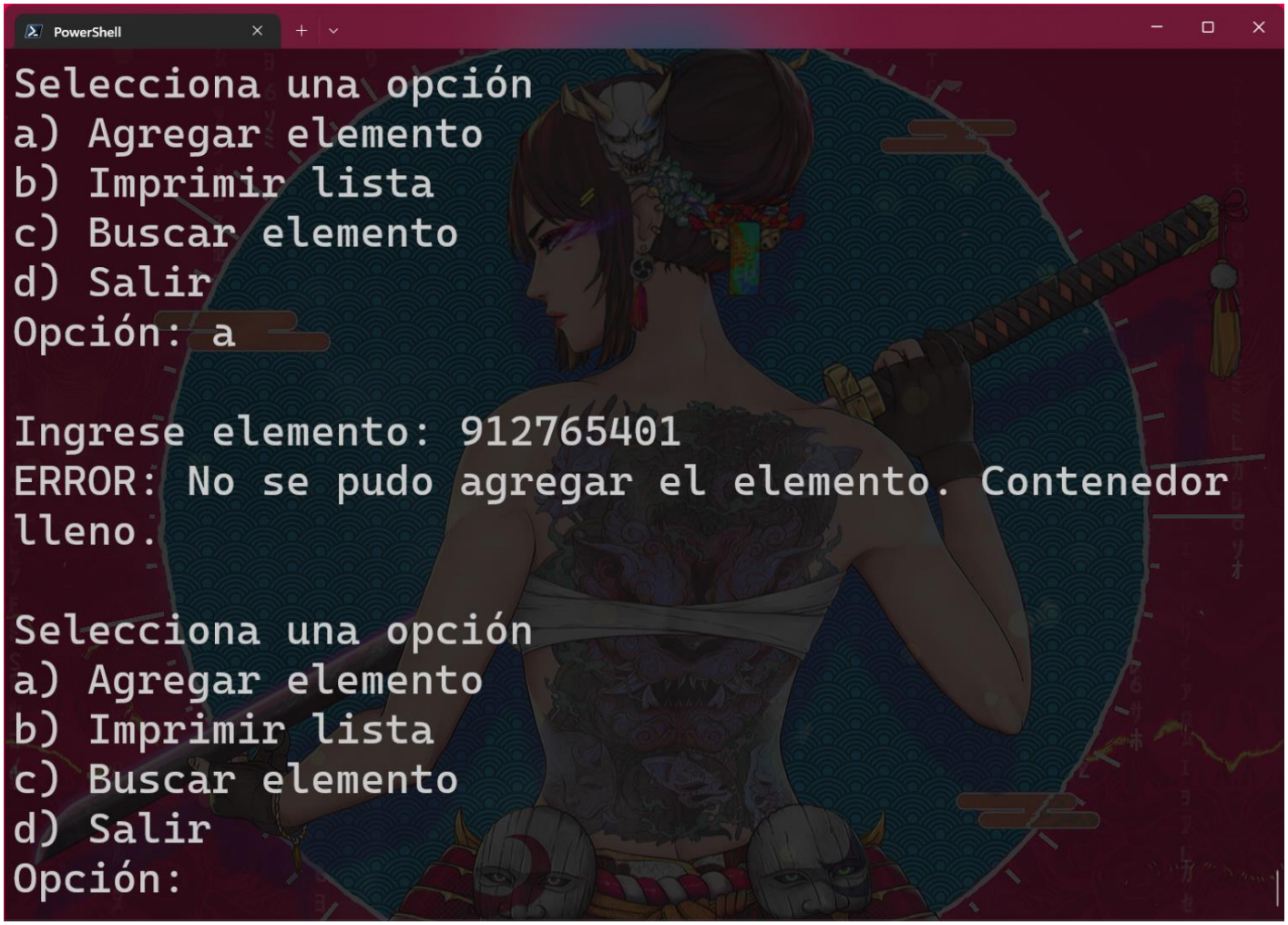
Buscando algunos elementos en la lista:



Agregando valores para llenar el contenedor:

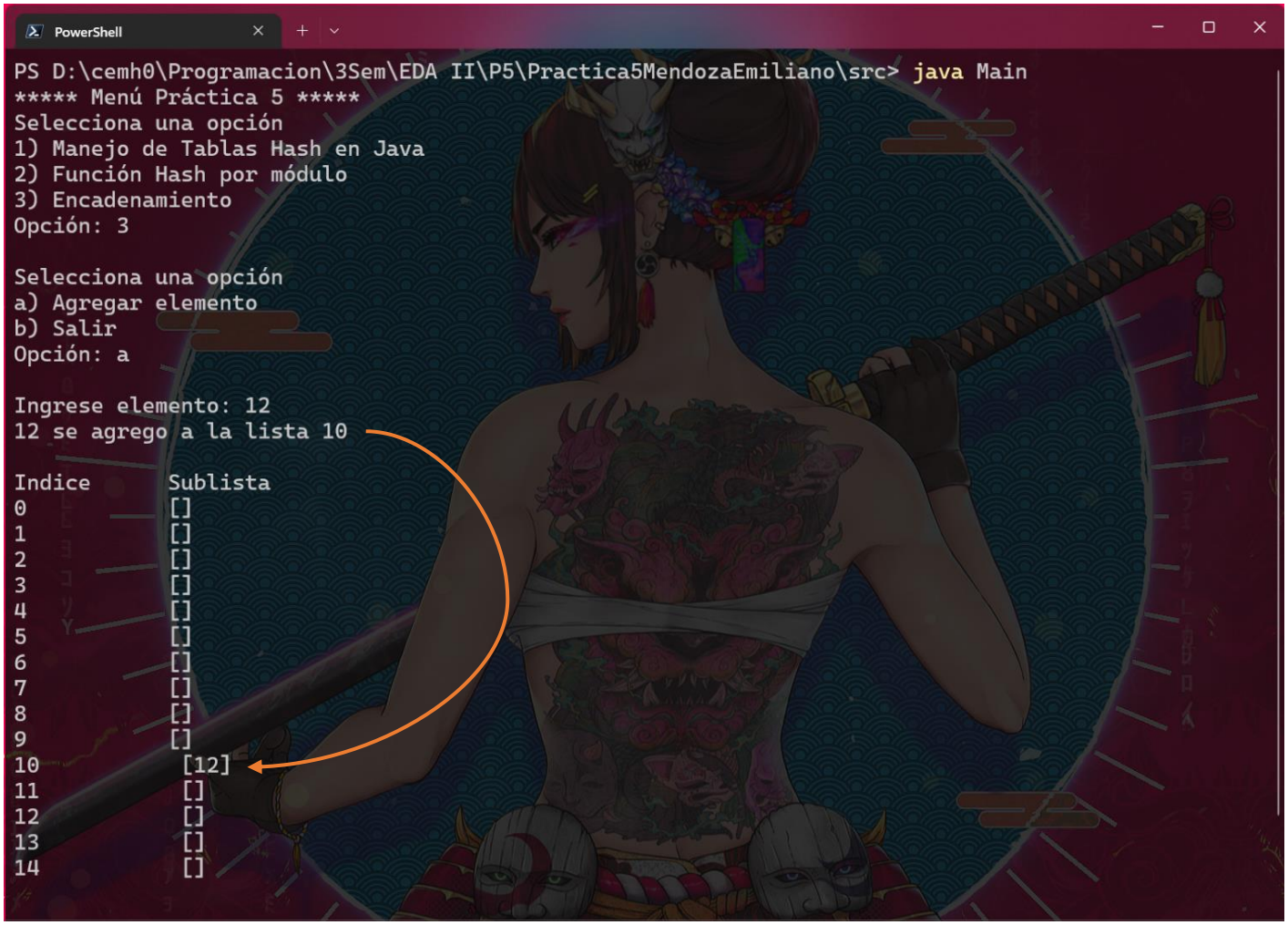


Agregando un elemento al contenedor lleno:

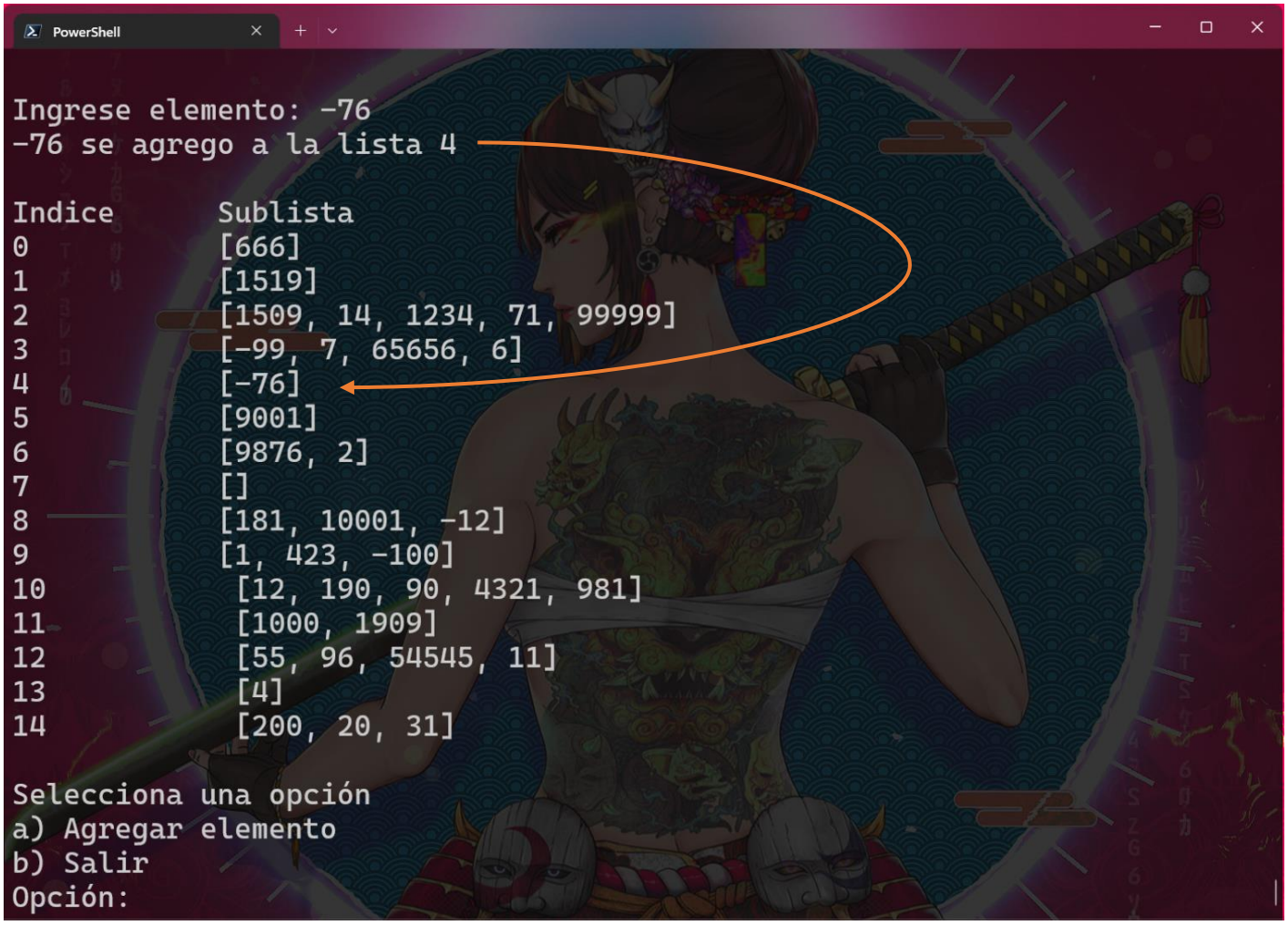


Por último, se ejecuta nuevamente el programa eligiendo la tercera opción del menú principal.

Ejecución en terminal



Agregando más valores:



Observaciones: Cada vez que el usuario ingresa un número se indica donde se almacena el valor y se imprime el estado actual de las listas.

Una vez probadas todas las opciones del menú, finaliza la ejecución del programa.

CONCLUSIONES

Algunos puntos que se concluyen de la primera actividad son los siguientes:

- El uso de Tablas Hash en Java puede manejarse con las implementaciones de la interfaz *Map<E>*, una de las cuales es *HashMap*; sin embargo, otra opción es *TreeMap.*
- Las tablas hash son colecciones de pares Clave-valor, y en ellas podemos agregar, buscar, modificar, eliminar (entre otras cosas) claves o valores.
- Después de utilizar sus métodos principales, se pudo observar que *HashMap* permite valores nulos y los elementos no se encuentran ordenados.

Para el segundo ejercicio se puede concluir que:

- Se puede simular el comportamiento de la búsqueda por transformación de claves, obteniendo claves del 0 al 14 para números de 9 dígitos usando la función hash por plegamiento y la función hash módulo.
- Las claves obtenidas en las funciones hash son localidades en el contenedor donde se almacenan los elementos.
- Para buscar un valor en el contenedor, basta con recuperar el valor o la clave del contenedor.
- El método de prueba lineal es una manera relativamente sencilla de solucionar los problemas de colisiones, sin embargo, es bastante probable que se den agrupamientos alrededor de ciertas clases, mientras que otras zonas del contenedor pueden permanecer vacías.

Además, al realizar el ejercicio 3 se llegó a las siguientes conclusiones:

- La lista de listas tiene un uso muy importante como método de solución de colisiones.
- En esta lista, cada elemento se convierte en el inicio de una lista ligada.
- En teoría, solo se agregan nuevos nodos cuando existen colisiones. En este caso, se agregaron nuevos nodos cuando se ingresó un nuevo valor al programa.
- Este método representa un uso eficiente de la memoria, y es una manera efectiva de resolver colisiones.

Dicho todo lo anterior, considero que se ha cumplido con cada uno de los objetivos planteados para el desarrollo de esta práctica dado que se conocieron y aplicaron algunos métodos para realizar la transformación de llaves, así como métodos para solucionar colisiones. Además, todo lo anterior se realizó con un enfoque orientado a objetos.

Como comentario final, considero que la búsqueda por transformación de llaves puede verse como una manera más “rápida” de hacer búsquedas, sin embargo, el compromiso en espacio de memoria es mayor al tener que crear un contenedor y claves para cada elemento. No hay un método de busca mejor que otro, todo dependerá de que aspecto se pretenda optimizar (tiempo de ejecución o uso de memoria).