



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

Grupo: 9

No de Práctica(s): 5

Integrante(s): Díaz Hernández Marcos Bryan

*No. de Equipo de
cómputo empleado:* Equipo Personal

No. de Lista o Brigada: 9

Semestre: 2021-1

Fecha de entrega: 04 de noviembre del 2020

Observaciones:

CALIFICACIÓN: _____

Objetivo de la practica

El estudiante conocerá e identificará las características necesarias para realizar búsquedas por transformación de llaves

Introducción

En esta practica se realizó el análisis de los ejercicios realizados en la practica 5, por ende fue necesario el realizar cada uno de los ejercicios, y a su vez realizar un diagrama breve de lo realizado para una mejor comprensión de lo elaborado, ya que realizo este reporte con la intención de regresar en un determinado momento y poder comprender lo que he realizado.

Ejercicios de la practica:

- Ejercicio 1.

El primer ejercicio, consistió en aprender a utilizar las tablas Hash, por medio de la interfaz Map, y de la clase HashMap, mas que nada era el saber implementar las tablas de forma sencilla y el utilizar algunos métodos sobre estas para ver el comportamiento que tienen.

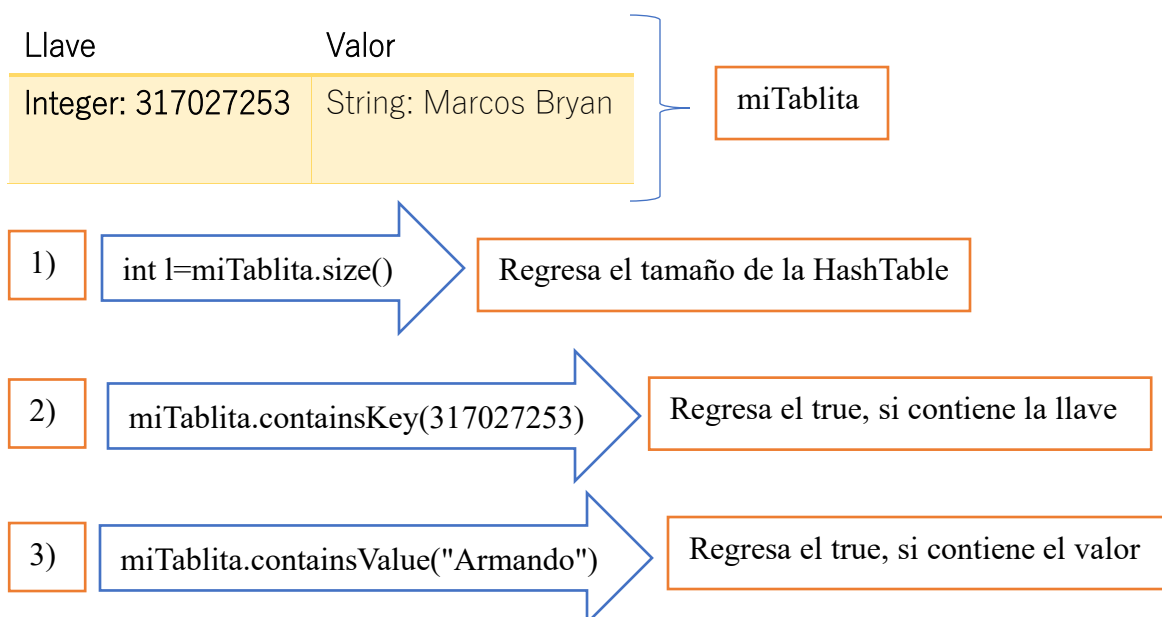
- Dificultades en el código

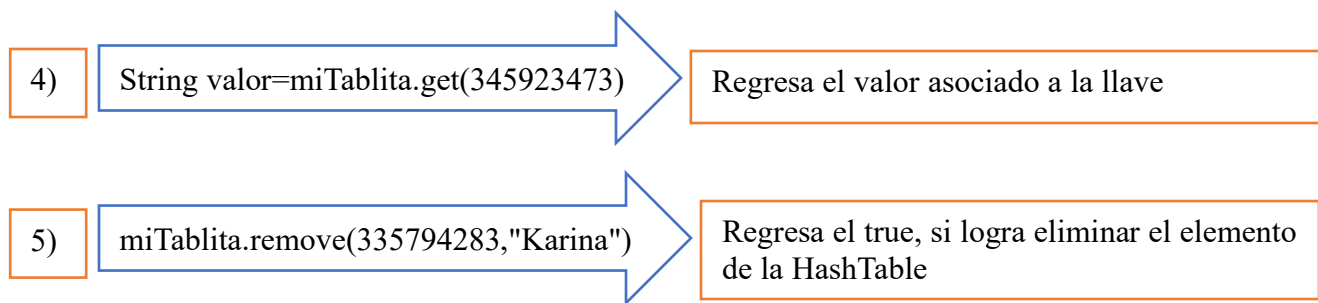
No hubo mucho con lo que poder lidiar, pero algo que si me hizo ruido fue la forma en que se declara, ya que se debe indicar el tipo de dato que corresponde a la llave y el tipo de dato del valor.

```
Map<Integer,String> miTablita = new HashMap<>();  
Map<Integer,String> miTablitaNueva = new HashMap<>();
```

Forma de declaración.

- Diagrama de funcionamiento:





- Relación con la teoría:

Con respecto a la teoría , vimos varias cosas relativas con las HashTable, pero en si no vimos el tema como tal en la materia pero en POO si vi algo relacionado con las colecciones de objetos, ya que están diseñadas para estos, así que no tuve mucho problema y con la API de Oracle, no tuve dificultad para buscar lo que hacía cada método y poder implementarlo a mi programa, la verdad es que es una herramienta bastante buena.

- Evidencia de implementación

The screenshot shows the NetBeans IDE interface. The main editor displays a Java file named `Practica5DiazMarcos.java` with the following code:

```
70 }
71
72 if(miTablita.equals(miTablitaNueva)){
73     System.out.println("Son iguales");
74 }else{
75     System.out.println("No son iguales");
76 }
77
78 miTablitaNueva=miTablita;
79
80 if(miTablita.equals(miTablitaNueva)){
81     System.out.println("Son iguales");
82 }else{
83     System.out.println("No son iguales");
84 }
85
86 String valor=miTablitaNueva.get(345923473);
87 System.out.println("Alumna:"+valor);
88 String valor1=miTablita.get(335794283);
89 System.out.println("Alumna:"+valor1);
90
```

The Output window at the bottom shows the execution results:

```
Practica5[DiazMarcos] (run) x Practica5[DiazMarcos] (run) #2 x
Alumna:Lynda
Alumna:Karina
Se elimino
Alumna expulsada:Lynda
Alumna:null
Alumna:null
Longitud:4
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Ejercicio 2:

El segundo ejercicio estuvo mas interesante, porque se tenía que elaborar una función Hash, o un acercado a esta, por medio de una nueva clase, que tuviera el método de Hash y que diera las posiciones para un arreglo de 15 posiciones, además se tenía que contemplar el caso de que hubiera colisiones, para esto se tenía que resolver por medio de prueba lineal.

- Dificultades en el código

Dentro del código todo estaba fluyendo bien hasta el punto de que tuve que comprobar las colisiones, ya que se indicaba en la práctica, que se tenía inicializar cada posición con un null, pero al utilizar los datos primitivos, pues estos no pueden ser asignados a null, y mucho menos compararse con estos, así que después de un rato de búsqueda encontré que se deben utilizar los wrappers que son cobertura o encapsulamiento de los tipos de datos primitivos que ayudan a que los datos puedan tomar o elaborar operaciones mas complejas, ya que se tratan como objetos. Una vez realizado esto, ya era posible comparar las posiciones con el null y determinar si la posición estaba ocupada.

```
if (compara == null) {
    return posicion;
} else {
    while (compara != null) {
```

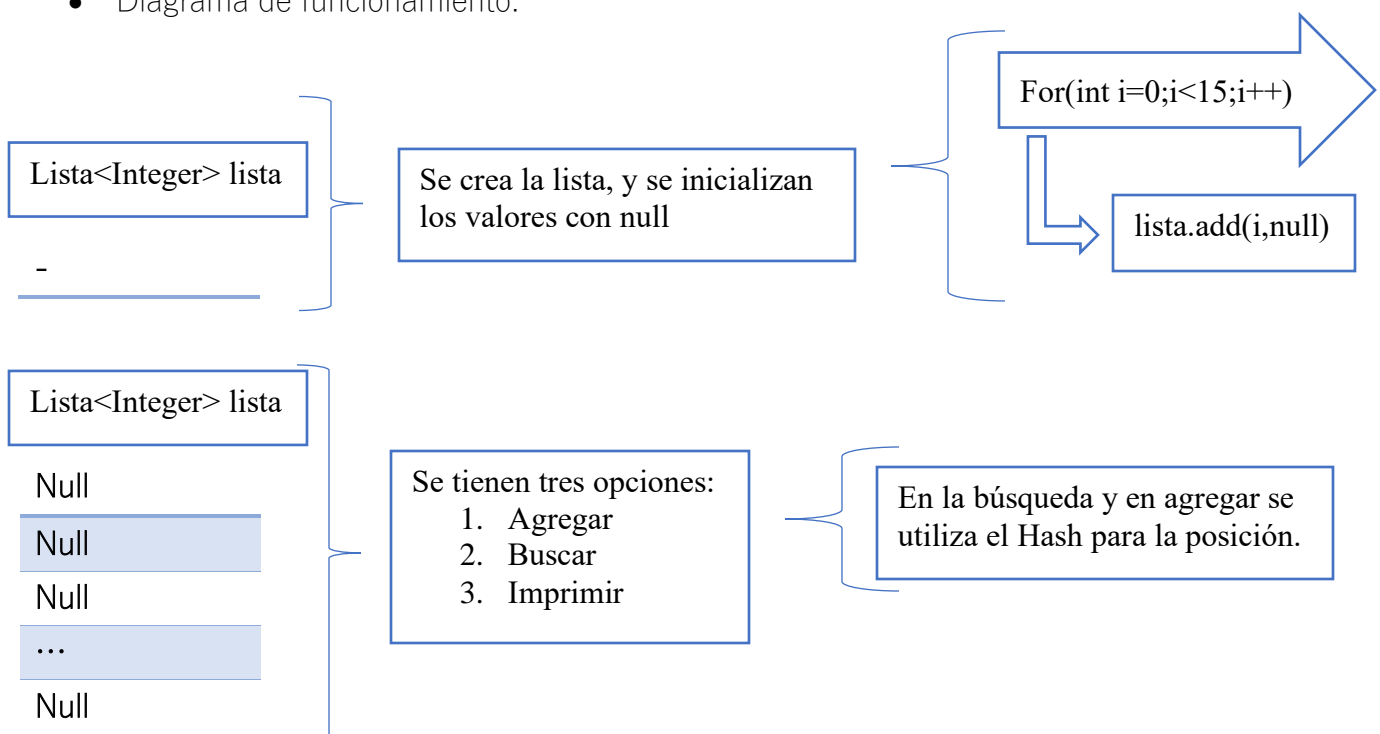
Comprobación de la posición.

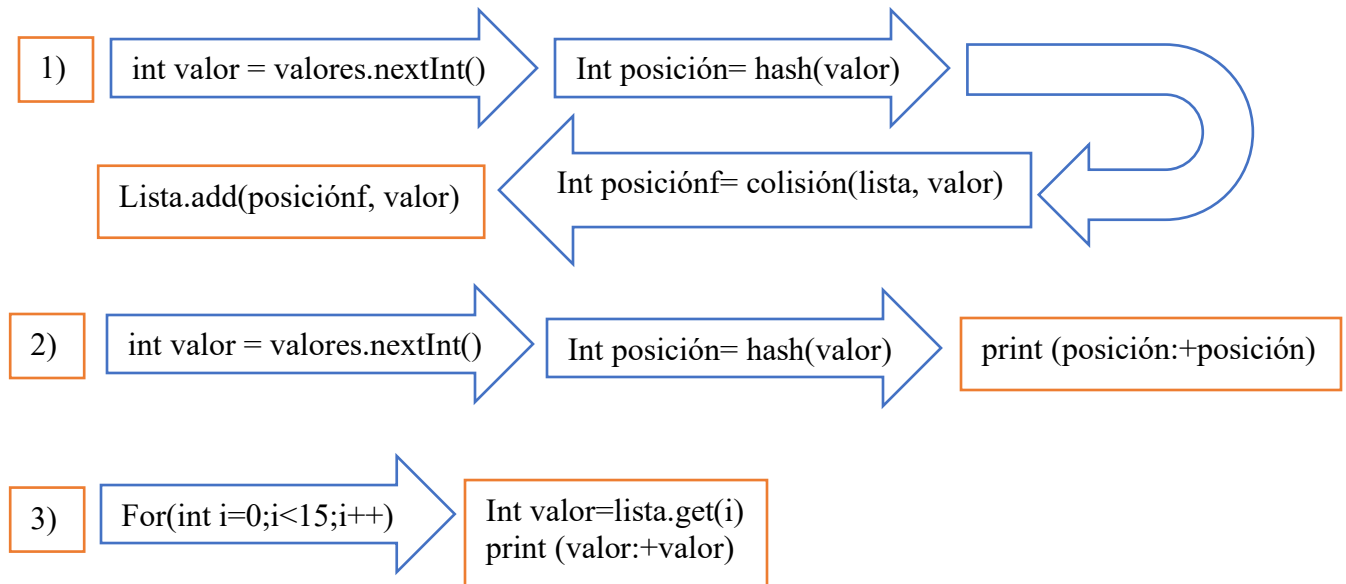
Dentro de otros métodos encontré varios puntos que considerar, como que el método del Hash debía revolver un valor entre [0,14] y que si no estaba el valor entre este rango se volviera a ejecutar para obtener una posición que estuviera dentro del rango, además de considerar que se usa un valor absoluto y solo convertir los negativos a positivos si fuera el caso.

```
public static int hash(int valor) {
    if (valor < 0) {
        valor = valor * -1;
    }
    if (valor > -1 && valor < 15) {
        return valor;
    } else {
        return hash(valor);
    }
}
```

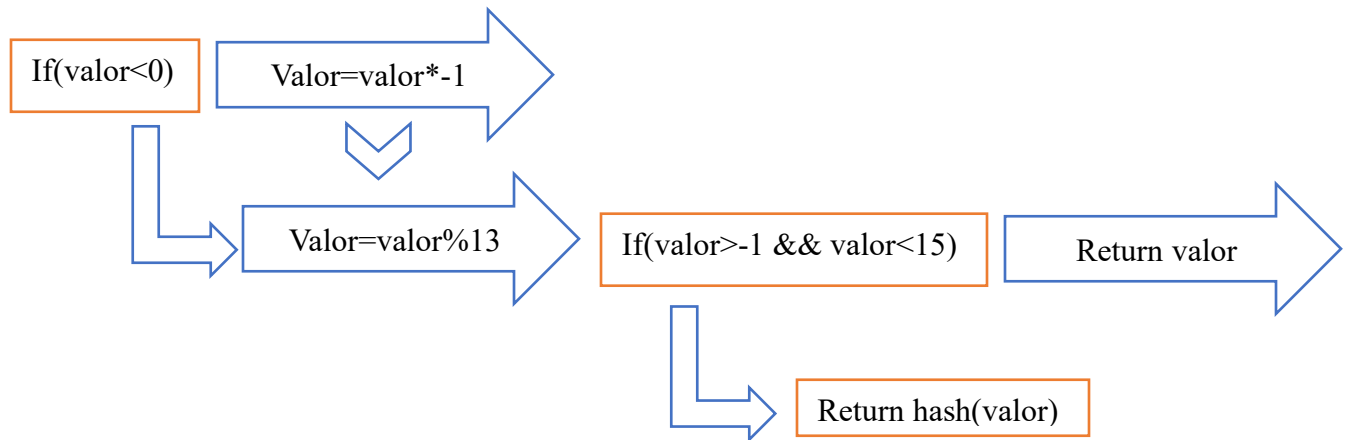
Consideraciones para el Hash.

- Diagrama de funcionamiento:

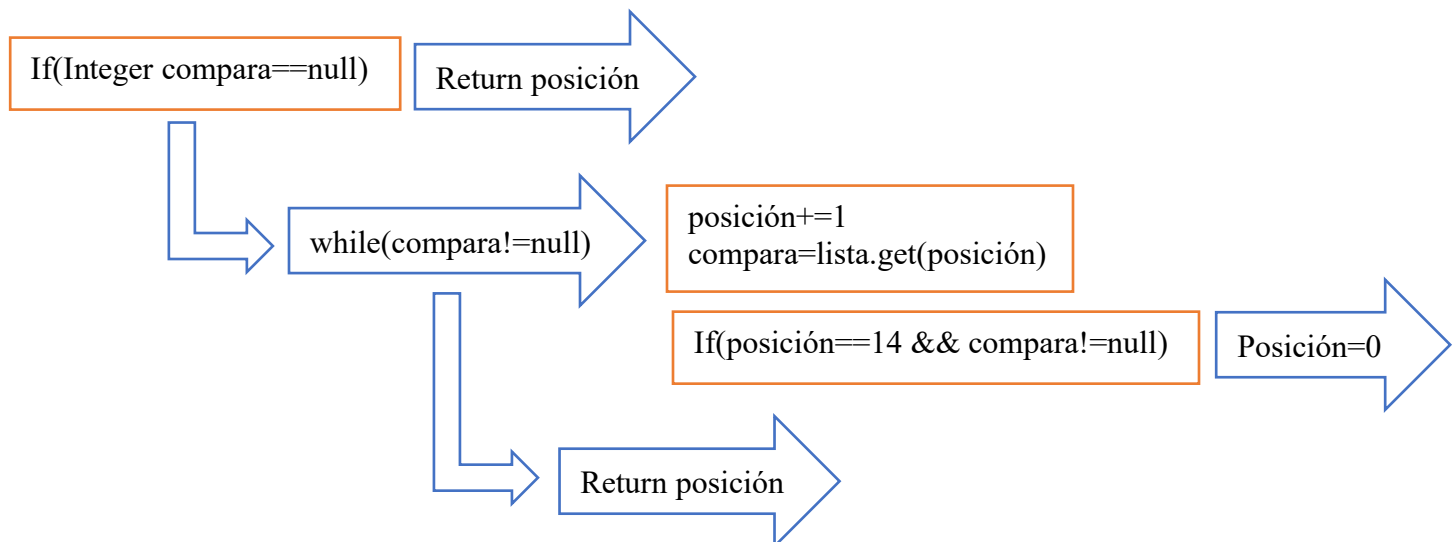




a) hash(valor)



b) colisión(lista, posición)



- Relación con la teoría:

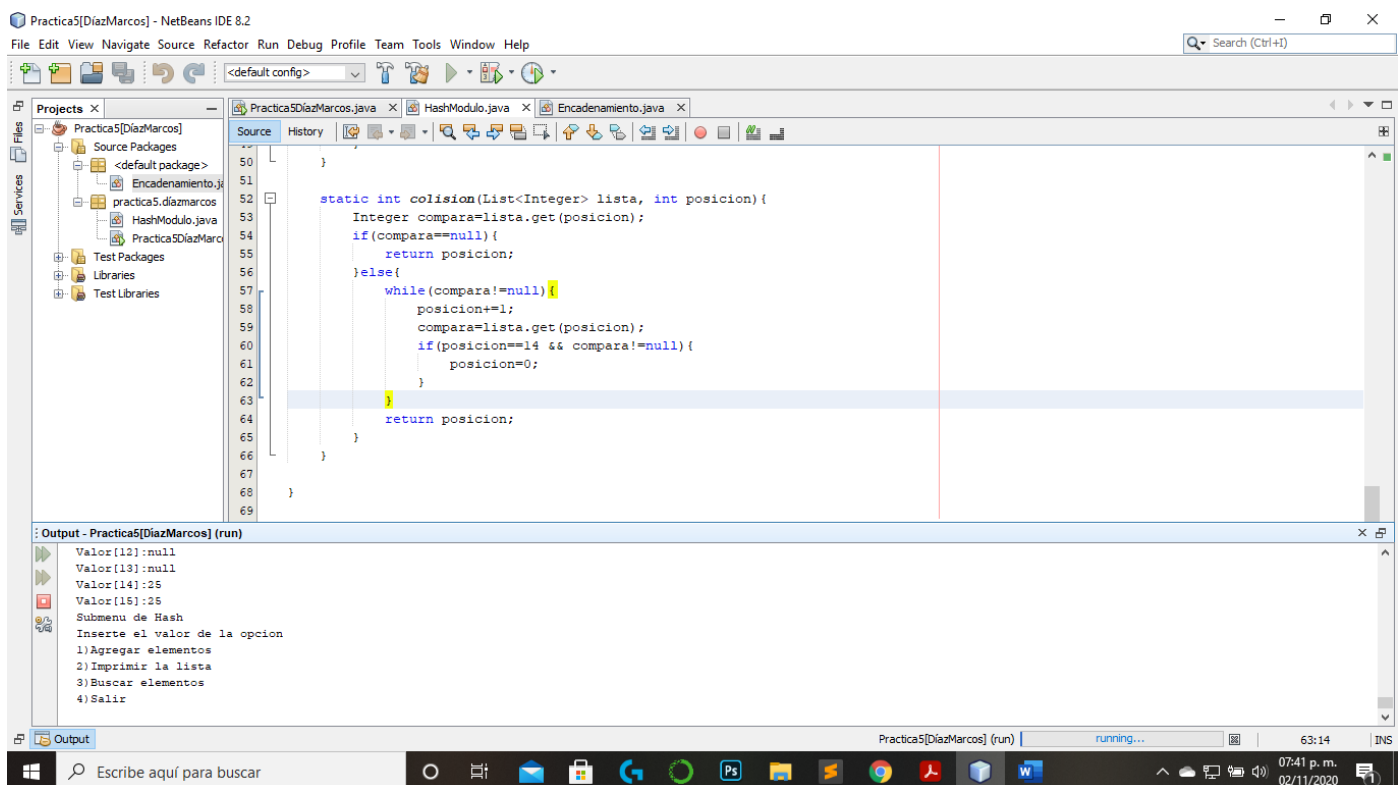
Con respecto a lo que se debe implementar en este ejercicio, se tiene la visión o contenido de la clase en línea de las funciones Hash, porque si no se tiene claro como se podría colisionar o como obtener el valor de la posición por medio del método, pues se pierde el motivo de usar la función Hash, sin embargo hay algo importante y creo que fue el uso de los wrappers, si bien no regrese el valor por medio del parse, se utiliza como el valor que se maneja en listas, que es Integer, porque se maneja como un objeto y de esta forma se puede iterar más fácilmente. Aunque dentro de la formalidad, no se debería dejar sin hacer el casteo con el parse para regresarlo a su valor original, o al tipo de dato, dentro de lo que he visto, pero en realidad eso no afecta el desarrollo del programa.

```
Integer compara=lista.get(posicion);  
if(compara==null){  
    return posicion;
```

Prueba del uso de wrapper.

Con respecto a lo demás, no hubo mucho nuevo, simplemente se consideraron más elementos para evitar que se accediera posiciones no deseadas o que se obtuvieran posiciones imposibles para una lista.

- Evidencia de implementación



- Ejercicio 3:

El tercer ejercicio pidió la implementación del encadenamiento, que se aplica cuando los valores colisionan en una misma posición, y para evitar el uso de los arreglos anidados que se vuelven un desperdicio en la memoria, se implementan las listas para cada posición, entonces el ejercicio pide eso, que cada vez que se añada un valor vaya directo a la lista correspondiente a la posición.

- Dificultades en el código

Esta parte es bastante conflictiva porque no resolví el ejercicio como se plantea en la práctica, pero comencé de esa forma, es decir que primero cree la lista de tipo lista, y después en cada posición iba a colocar una lista, y para acceder a los valores de la cada lista, pues la posición de cada una me ayudaría a poder agregar los valores.

Pero me dije, se supone que se pidió el uso de las HashTable en el primer ejercicio para saber como se usan, y pues como vi que el ejercicio se prestaba para la solución por medio de estas, ya que las posiciones son las llaves, y el valor es cada lista, decidí usarlas.

```
static void integrar(Map<Integer,List<Integer>> lista){  
    for(int i=0;i<15;i++){
```

Prueba de la implementación.

Ya que sabía como utilizarlas, pues no estuvo difícil hacerlo por medio de estas, ya que solo coloque el tipo de valor de la lista, como List<Integer> y para inicializar o colocar dentro de cada key una lista, solo use un for del [0,14], para añadir una lista cada key.

```
for(int i=0;i<15;i++){  
    List<Integer> valor =new LinkedList<>();  
    lista.put(i, valor);
```

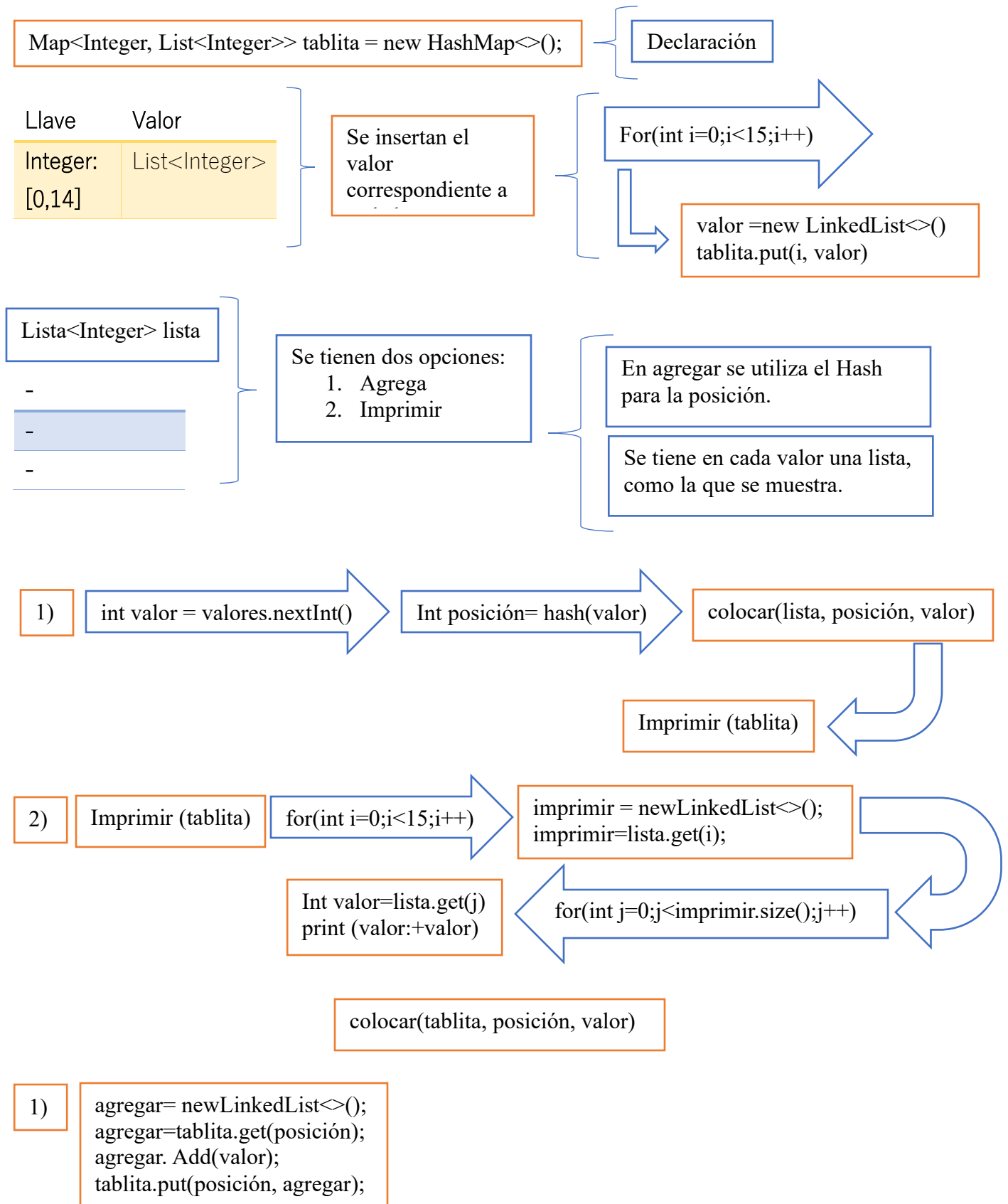
Asignación de las llaves y de las listas.

Después, para obtener el valor de la posición, use como función aleatoria la misma del Hash por valor absoluto, y como las colisiones se evitan al asignar una lista a cada valor de posición posible se manda directo a colocar al valor dentro de la lista correspondiente, para esto la key es la posición y con esta se obtiene la lista y después se le añade el valor, y se vuelve a regresar.

```
List<Integer> agregar =new LinkedList<>();  
agregar=lista.get(posicion);  
agregar.add(valor);  
lista.put(posicion, agregar);
```

Como se añaden los valores a las listas de cada posición.

- Diagrama de funcionamiento:



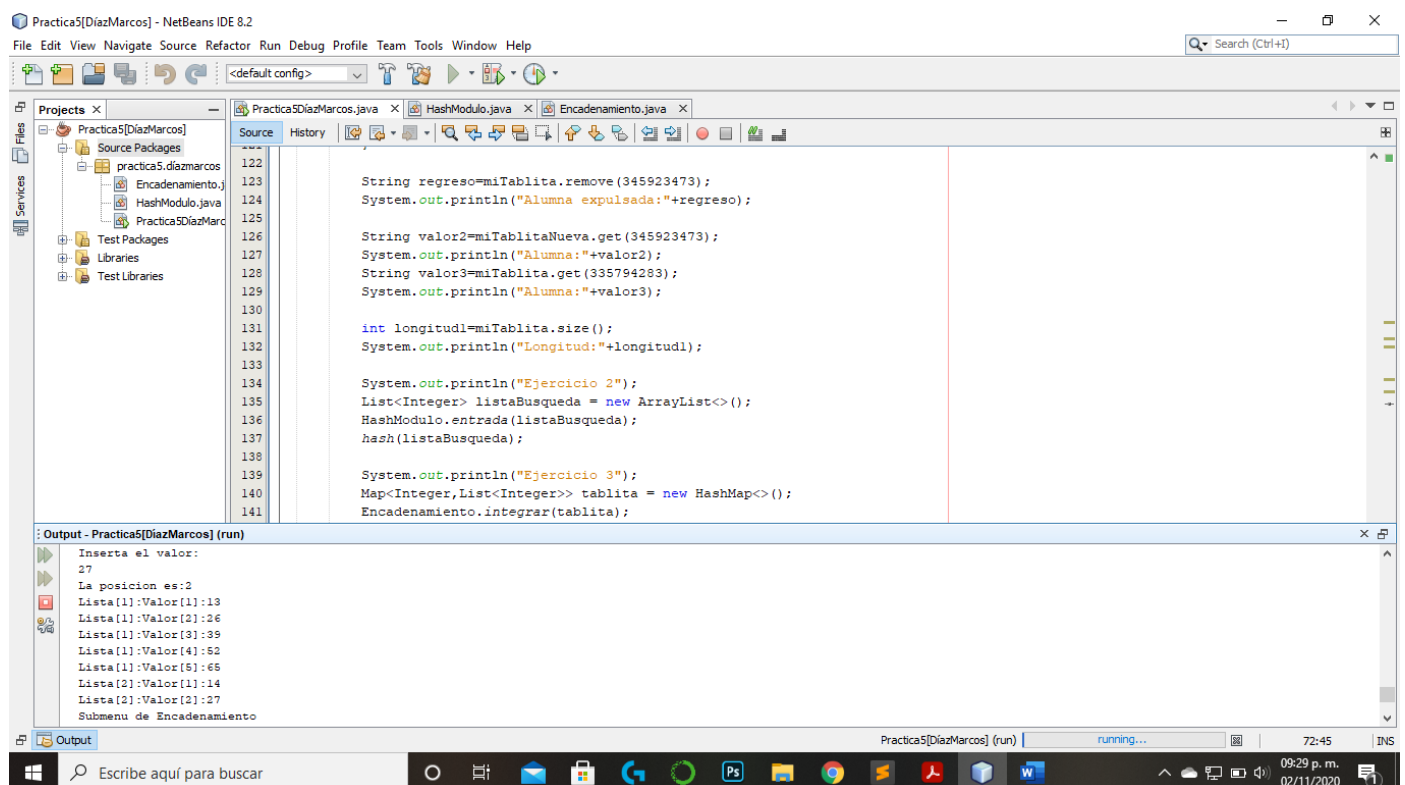
- Relación con la teoría:

Con respecto a la teoría, creo que lo hice la versión mas sencilla, aunque el análisis de la versión que pedía la práctica, me llevo a esta solución además que se plantaba con la intención de que se cumpliera con la forma del encadenamiento, de lo contrario no lo hubiese hecho de la forma en que lo hice.

Por otro lado, la teoría que se vio para poder realizar la práctica estuvo bastante relacionada, por no decir que si quedo claro lo visto en clase se podía resolver rápido los ejercicios, ya que Java se presta bastante a la reutilización del código, entonces se podían volver a utilizar el método del Hash, y solo modificar los otros métodos a modo de que se aplicaran al caso de los valores, y el tipo de colección que se estaba usando.

Otro aspecto que esta curioso es que en las HashTable no es posible iterar, a menos que se use el Enumeration como interfaz, pero el principio de esta es que se obtienen las llaves y después se van obteniendo los valores correspondientes, pero al colocar yo las llaves y saber cual es el valor que estas deben tener es como pude iterar sobre estas sin el Enumeration, esto me acaba de llegar después de haber realizado el ejercicio, y tiene bastante sentido con respecto a la forma en que fue implementado,

- Evidencia de implementación



```
Practica5[DíazMarcos] - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Projects
  Practica5[DíazMarcos]
    Source Packages
      practica5.díazmarcos
        Encadenamiento.java
        HashModulo.java
        Practica5DíazMarcos.java
    Test Packages
    Libraries
    Test Libraries
Source
  122
  123
  124
  125
  126
  127
  128
  129
  130
  131
  132
  133
  134
  135
  136
  137
  138
  139
  140
  141
String regreso=miTablita.remove(345923473);
System.out.println("Alumna expulsada:"+regreso);

String valor2=miTablitaNueva.get(345923473);
System.out.println("Alumna:"+valor2);
String valor3=miTablita.get(335794283);
System.out.println("Alumna:"+valor3);

int longitudl=miTablita.size();
System.out.println("Longitud:"+longitudl);

System.out.println("Ejercicio 2");
List<Integer> listaBusqueda = new ArrayList<>();
HashModulo.entrada(listaBusqueda);
hash(listaBusqueda);

System.out.println("Ejercicio 3");
Map<Integer,List<Integer>> tablita = new HashMap<>();
Encadenamiento.integrar(tablita);

Output - Practica5[DíazMarcos] (run)
Inserta el valor:
27
La posición es:2
Lista(1):Valor(1):13
Lista(1):Valor(2):26
Lista(1):Valor(3):39
Lista(1):Valor(4):62
Lista(1):Valor(5):65
Lista(2):Valor(1):14
Lista(2):Valor(2):27
Submenu de Encadenamiento

Practica5[DíazMarcos] (run) | running... | 72:45 | INS
Escribe aquí para buscar
```

Conclusiones

Durante el proceso de la elaboración de la presente practica y reporte, poco a poco fui entendiendo las partes esenciales de la inserción de elementos y la búsqueda de estos, por medio de una función Hash, que es una función que regresa un valor que corresponde a la posición del elemento dentro de una lista, de una colección, donde lo principal es tener la función para poder llevar a cabo la transformación del valor y obtener una llave-posición, además de tener las correspondientes consideraciones para que en caso de colisiones se pueda determinar una posición adecuada.

Por último se tiene que este tipo de funciones Hash permiten de igual forma buscar valores y determinar, si se encuentran, ya que se obtiene la posición y se busca directo en esta el valor, por lo que se puede concluir que se cumplió el objetivo de la práctica.