Chapter 4. Collection of objects. Maps

Metodología y Desarrollo de Programas

Tablas de dispersión

- El objetivo de toda estructuras de datos es proporcionar una búsqueda lo más eficiente posible (igualmente con las operaciones de inserción y borrado)
- La búsqueda
  - Búsqueda Secuencial: T(n)=n necesita recorrer todo la estructurada de datos
    - ✓ Si es un array todo el array
    - ✓ Si es una lista, conjunto, pila, … todo los nodos de la ED
  - Búsqueda Binaria (ordenada):

MaxVector	Búsqueda Secuencial O(n)	Búsqueda Binaria O(log n)
7	7	3
100	100	7
1.000	1.000	10
1.000.000	1.000.000	20

Tablas de dispersión

- Existen otros métodos que permite aumentar la velocidad de búsqueda, sin necesitar, que los datos estén ordenados → Transformación de claves (clave-dirección) o hashing
- La idea consiste en acceder directamente a los datos de la búsqueda a partir de su **clave**, es decir, consiste en acceder a la información a partir de su la clave, la cual es la dirección dentro del array
- Por ejemplo:
  - Supongamos que queremos almacenar los estudiantes de EDI (47). Para ello asignaríamos a cada estudiante un identificador único (comprendido entre 1 -47), de modo que el estudiante identificado con el valor 1 estaría siempre en la posición 1 (no puede haber identificadores repetidos)
    - ✓ Solución eficiente: t(n)=1

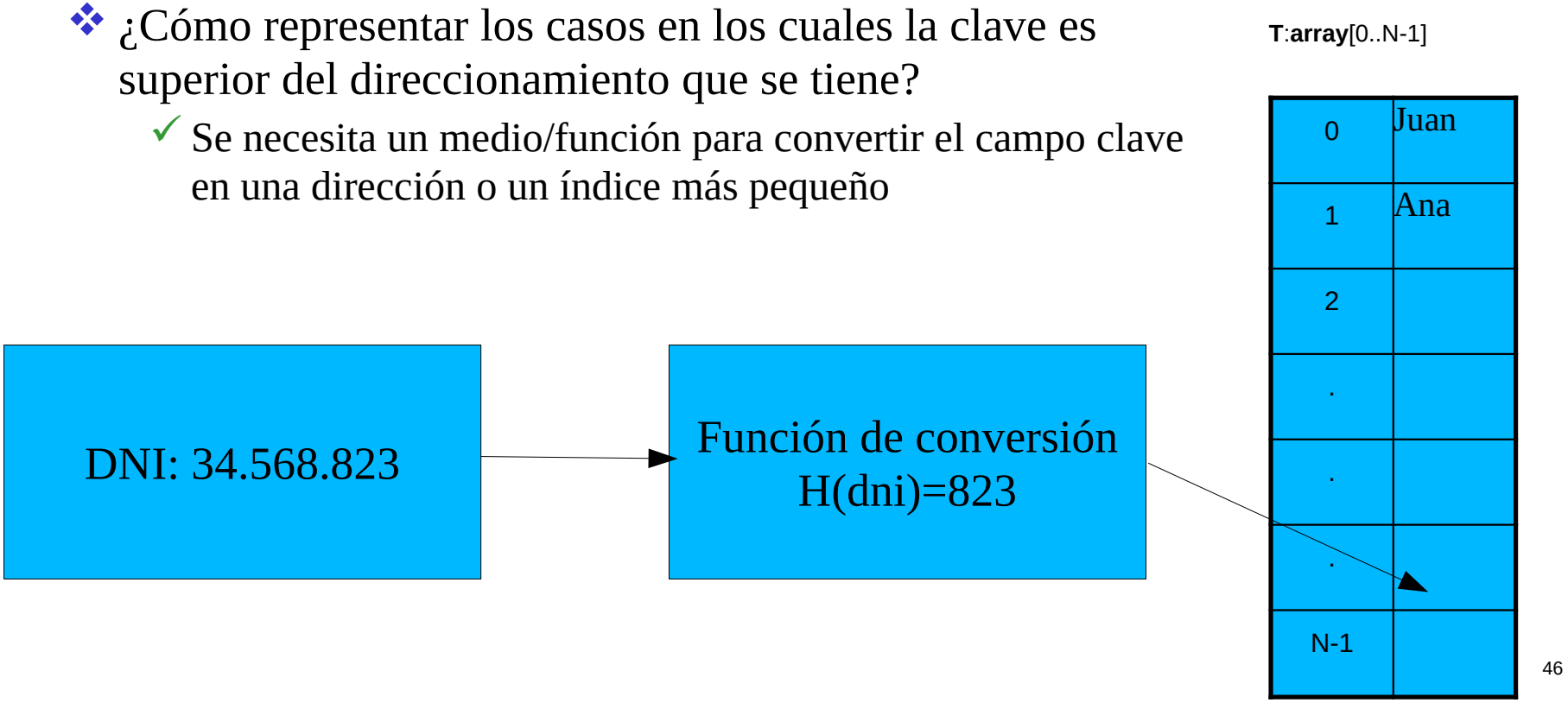
T:**array**[0..N-1]

0	Juan
1	Ana
2	
.	
.	
.	
N-1	

46

Tablas de dispersión

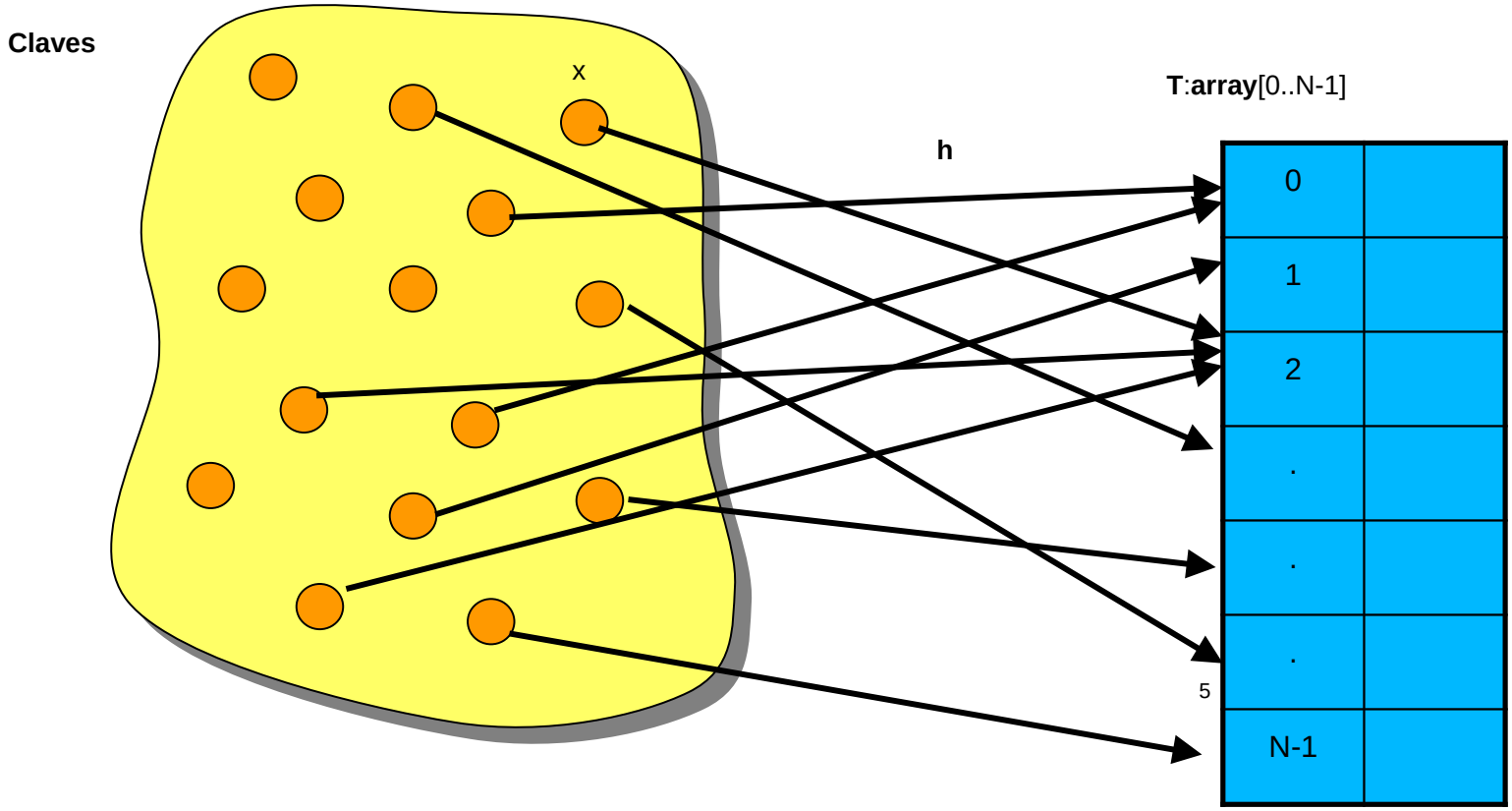
- Supongamos que queremos representar la clave como el número del DNI, de la seguridad social o el expediente
  - Necesitaríamos un array de dimensiones muy difícil de gestionar
  - ¿Cómo representar los casos en los cuales la clave es superior del direccionamiento que se tiene?
    - ✓ Se necesita un medio/función para convertir el campo clave en una dirección o un índice más pequeño



46

Tablas de dispersión

- Idea:
  - Reservar un tamaño fijo, un array T con N posiciones (0, ..., N-1).
  - Dada una clave x (sea del tipo que sea) calcular la posición donde colocarlo, mediante una función h.

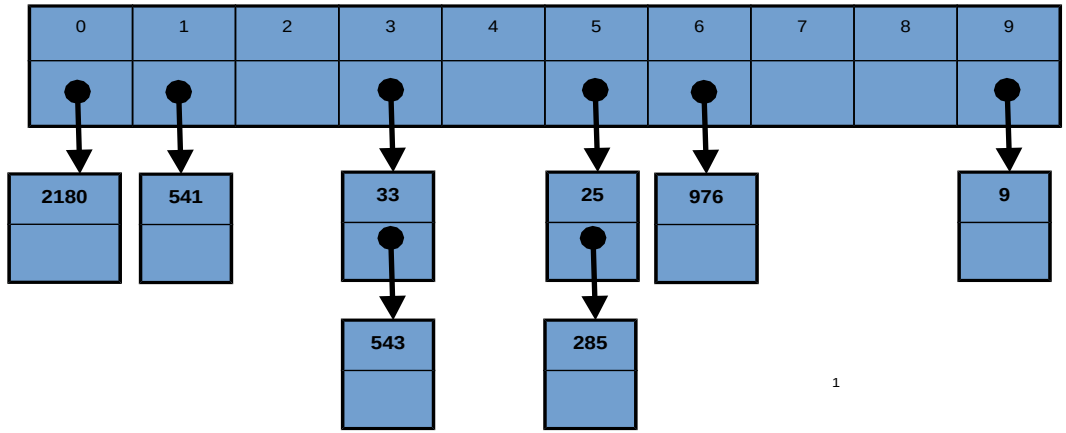


Tablas de dispersión

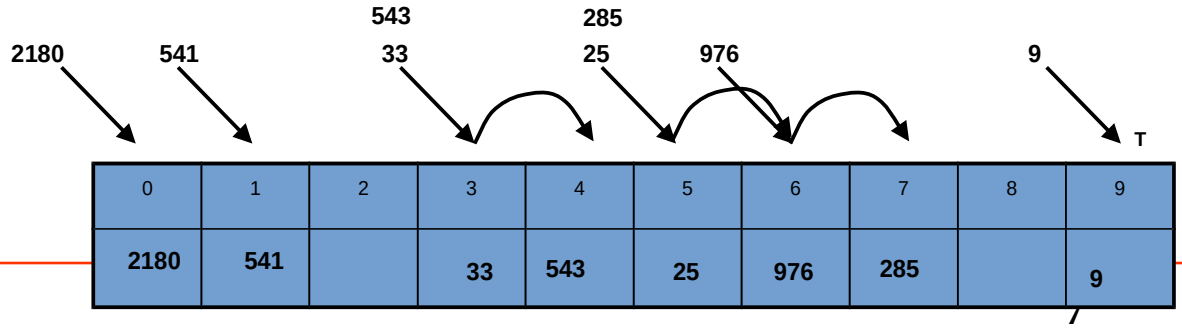
- Inserción
  - Para almacenar un elemento en la tabla hash se ha de convertir su clave a un número. Esto se consigue aplicando la función resumen (hash) a la clave del elemento.
  - El resultado de la función resumen ha de mapearse al espacio de direcciones del vector que se emplea como soporte
  - El elemento se almacena en la posición de la tabla obtenido en el paso anterior.
  - Si en la posición de la tabla ya había otro elemento, se ha producido una colisión → Ya se verá que se hace.
- Búsqueda
  - Para recuperar los datos, es necesario únicamente conocer la clave del elemento, a la cual se le aplica la función resumen.
  - El valor obtenido se mapea al espacio de direcciones de la tabla.
  - Si el elemento existente en la posición indicada en el paso anterior tiene la misma clave que la empleada en la búsqueda, entonces es el deseado.

Funciones de dispersión o hash

- ¿Qué ocurre si para dos elementos distintos x e y, tienen la **misma clave**?
  - Al tener la misma clave, se consideran iguales y se sobreescribe.
- ¿Qué ocurre si para dos elementos distintos x e y, ocurre que h(x) = h(y)?
  - Definición: Si (x ≠ y) Y (h(x) = h(y)) entonces se dice que x e y son sinónimos.
  - Los distintos métodos de dispersión difieren en el tratamiento de los sinónimos.
- Colisión: Tipos de dispersión (hashing):
  - Dispersión abierta.



  - Dispersión cerrada.



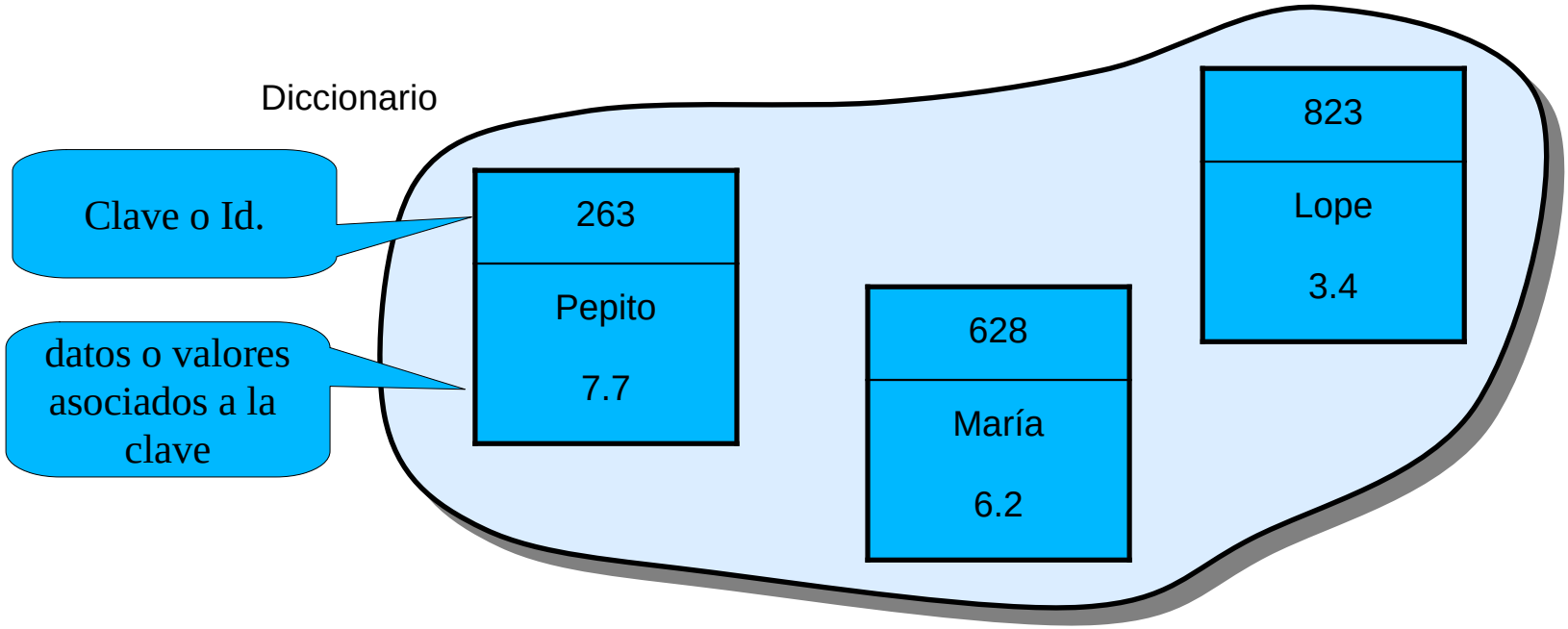
TAD Diccionario

- En la vida real existen muchos ejemplos en los que se utiliza el concepto de conjunto (colección de elementos no repetidos) pero que no se puede permitir tener una complejidad T(n)=n a la hora de insertar
  - Ejemplos. Agenda electrónica, diccionario de sinónimos, base de datos de empleados, notas de alumnos, etc.

- Todos estos casos tienen una particularidad: el **valor/dato** que se quiere almacenar **tiene asociada una clave**:
  - Agenda: Tiene un identificador de usuario o el número de teléfono
  - Notas de alumnos: tienen un identificador como el dni o el num. Expediente
  - ...
- En todos estos casos el **comportamiento** de estas estructuras de datos será **en base a la clave o identificador de los datos a almacenar**.

TAD Diccionario

- Objetos vistos por claves



TAD Diccionario

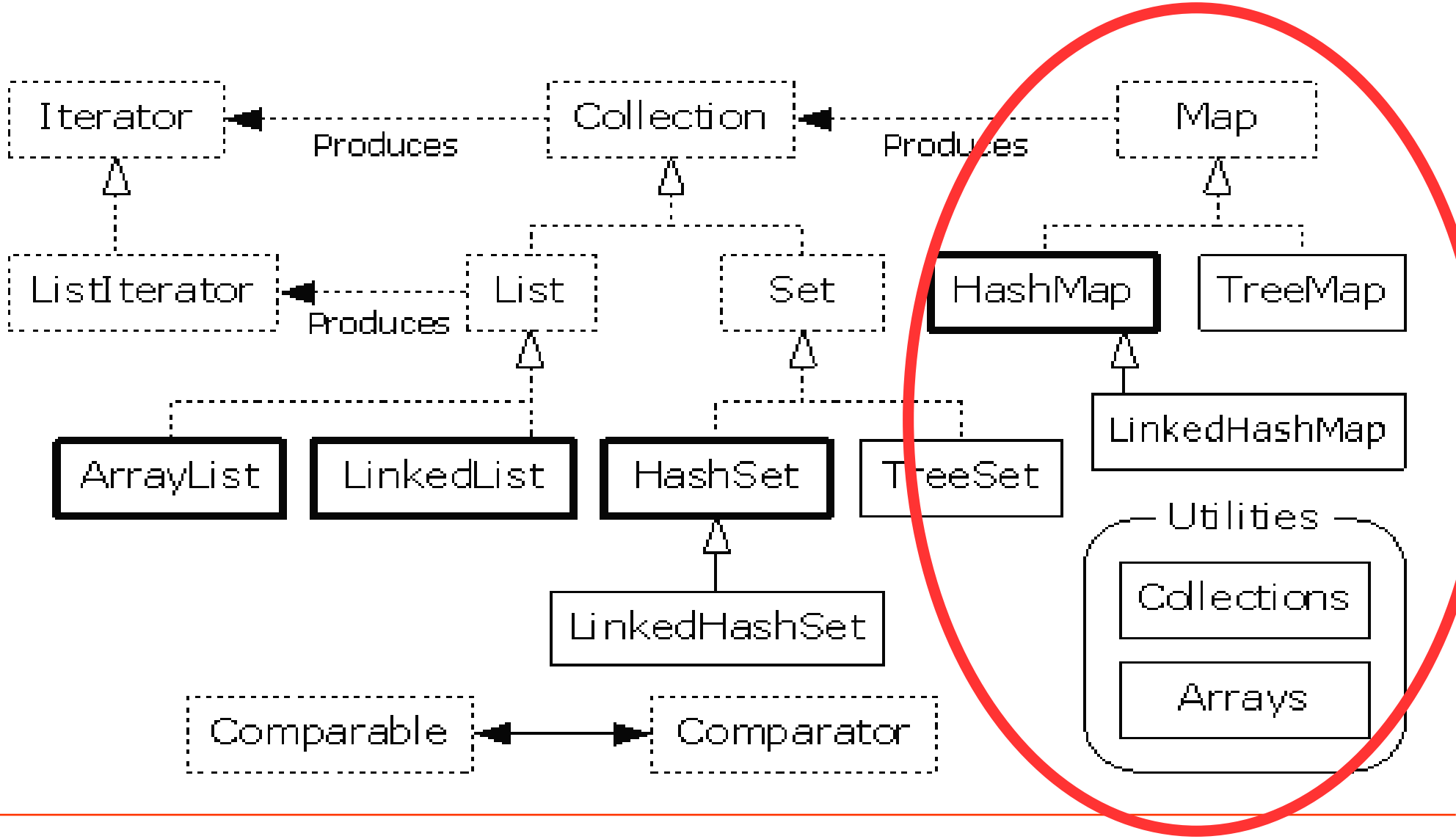
Diccionario
Diccionario() inserta(K clave, Object valor): void borra(K clave): void pertenece(K clave): boolean consulta(K clave): Object esVacio(): boolean

TAD Diccionario

- Independiente de la implementación que se escoja se tendría un interface común que podría ser:

```
public interface Diccionario {
  public void inserta(K clave, Object valor); // inserta o modifica
  public void borra(K clave);
  public boolean pertenece(K clave);
  public Object consulta(K clave) throws ClaveNoInsertadaExcepcion;
  public boolean esVacio();
}
```

Collections' Outline



Map functionality

- To store a sequence of objects using other criterion non-based on its position.
- It is called as a **map**, a **dictionary**, or an **associative array**.
- Maintains key-value associations (pairs) so you can look up a value using a key.
- **Methods**
  - Object **put**(Object key, Object value): To add an object
  - Object **remove**(Object key): To remove an object
  - Object **get**(Object key): To get an object
  - **ContainsKey:** Check if a key exists
  - **ContainsValue:** Check if a value exists
  - **isEmpty**,**size, etc**

- A map object can only store a value for each key. If we want to store more than one value, it has to define as a list of value

Map usage example

- HashMap's Example:
  - In order to retrieve an object, its key is necessary (.get(key)) as well as to store in the map.
  - If a key exists and a put operation is made, its object is overwritten.

```
HashMap<String,Alumno> hashMap = new HashMap<String,Alumno>();
-----------
hashMap.put(“51”, new Alumno(“Luis”,”51”,...));
hashMap.put("102",new Alumno(“Juan”,”102”,....));
if (!hashMap.containsKey(“404”))
        hashMap.put("404",new Alumno(“Ana”,”404”,.....));
-----
Alumno aux=(Alumno) hashMap.get(“404”);
```

Map functionality

Map (interface)	Maintains key-value associations (pairs) so you can look up a value using a key.
HashMap*	Implementation based on a hash table. (Use this instead of **Hashtable**.) Provides constant-time performance for inserting and locating pairs. Performance can be adjusted via constructors that allow you to set the *capacity* and *load factor* of the hash table.
LinkedHashMap (JDK 1.4)	Like a **HashMap**, but when you iterate through it, you get the pairs in insertion order, or in least-recently-used (LRU) order. Only slightly slower than a **HashMap**, except when iterating, where it is faster due to the linked list used to maintain the internal ordering.
TreeMap	Implementation based on a red-black tree. When you view the keys or the pairs, they will be in sorted order (determined by **Comparable** or **Comparator**, discussed later). The point of a **TreeMap** is that you get the results in sorted order. **TreeMap** is the only **Map** with the **subMap( )** method, which allows you to return a portion of the tree.
WeakHashMap	A map of *weak keys* that allow objects referred to by the map to be released; designed to solve certain types of problems. If no references outside the map are held to a particular key, it may be garbage collected.
IdentityHashMap (JDK 1.4)	A hash map that uses == instead of **equals( )** to compare keys. Only for solving special types of problems; not for general use.

Map functionality

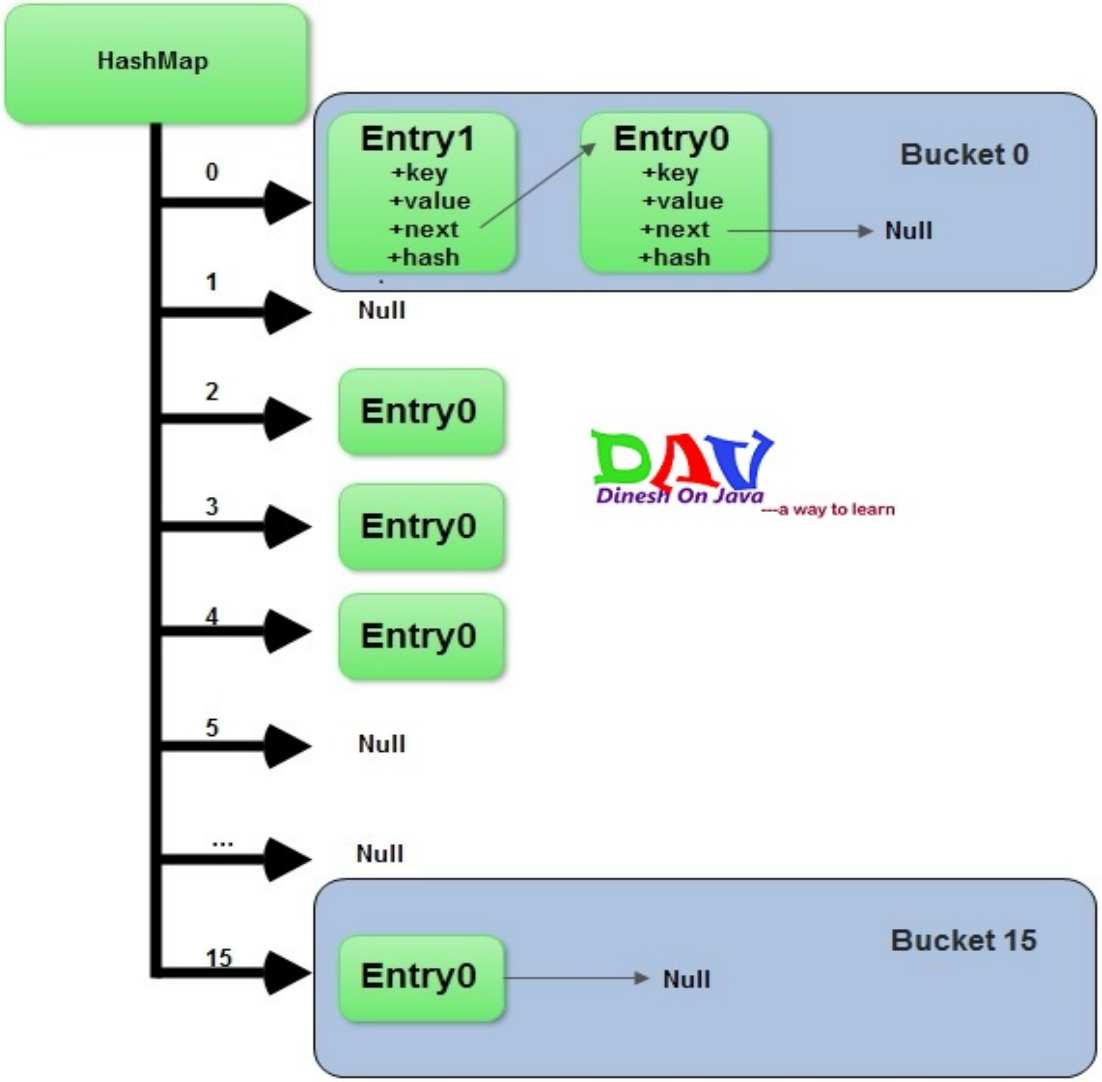
- How does java Hashmap work internally ?
  - HashMap is **an array of Entry objects**:

```
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    Entry<K,V> next;
    final int hash;
    ...
}
```

  - Each Entry object represents **key-value** pair.
  - **Field next** refers to other Entry object if a bucket has more than 1 Entry.
  - Sometimes it might happen that hashCodes for 2 different objects are the same → **Open addressing** (**dispersión abierta**)
    - ✓ In this case 2 objects will be saved in one bucket and will be presented as LinkedList. The entry point is more recently added object.
    - ✓ This object refers to other object with next field and so one. Last entry refers to null.

Map functionality

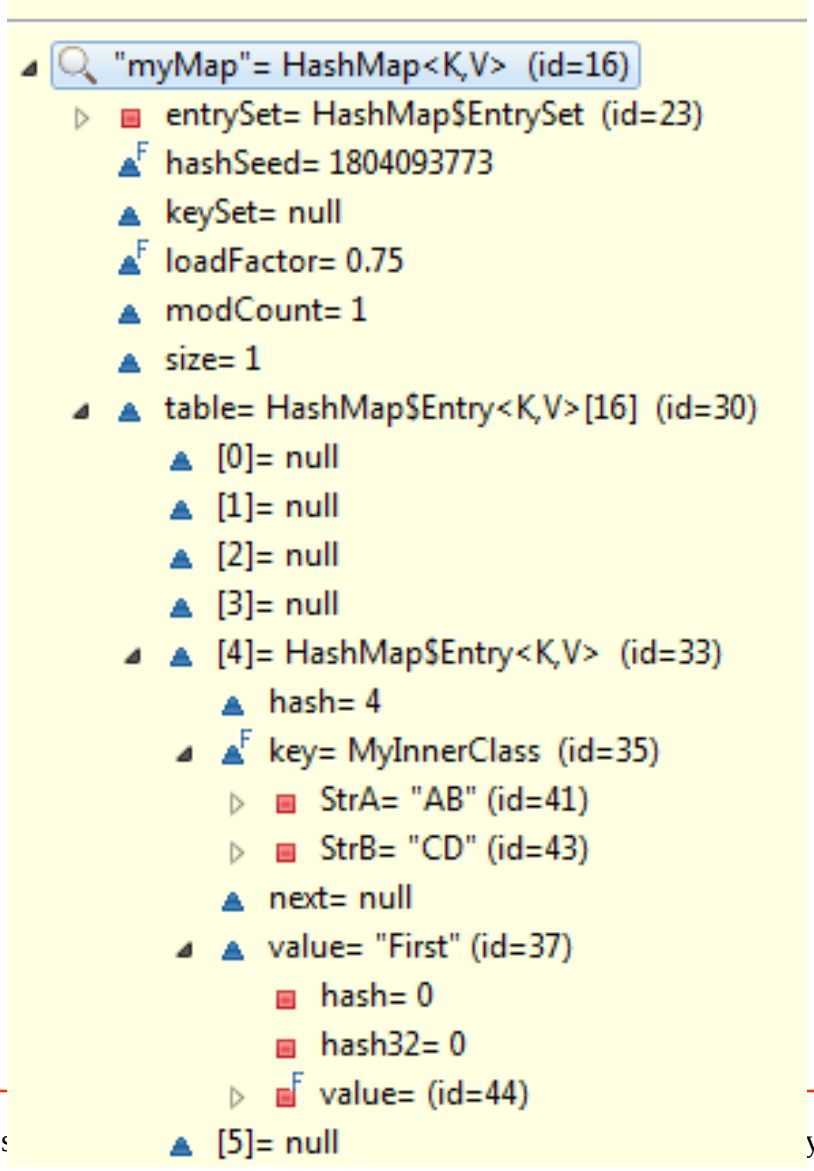
- How does java Hashmap work internally ?



Obtained from: http://www.dineshonjava.com/2013/06/how-does-java-hashmap-work-internally.html

Map functionality

- How does java Hashmap work internally ?



Map functionality

- How does java Hashmap work internally ?
  - Adding a new key-value pair
    - ✓ Calculate hashcode for the key
    - ✓ Calculate position hash % (arrayLength-1)) where element should be placed(bucket number)
    - ✓ If you try to add a value with a key which has already been saved in HashMap, then value gets overwritten.
    - ✓ Otherwise element is added to the bucket. If bucket has already at least one element - a new one is gets added and placed in the first position in the bucket. Its next field refers to the old element.
  - Deletion:
    - ✓ Calculate hashcode for the given key
    - ✓ Calculate bucket number (hash % (arrayLength-1))
    - ✓ Get a reference to the first Entry object in the bucket and by means of equals method iterate over all entries in the given bucket. Eventually we will find correct Entry. If desired element is not found - return null

Map usage example

- Maps do not provide an iterator() method as do Lists and Sets.
- A Set of either keys (keySet()) or key-value Map .Entry elements (entrySet()) can be obtained from the Map, and one can iterate over that.
  - .getKey(): It obtains the key for the selected object
  - .getValue(): It obtains the object from the selected key

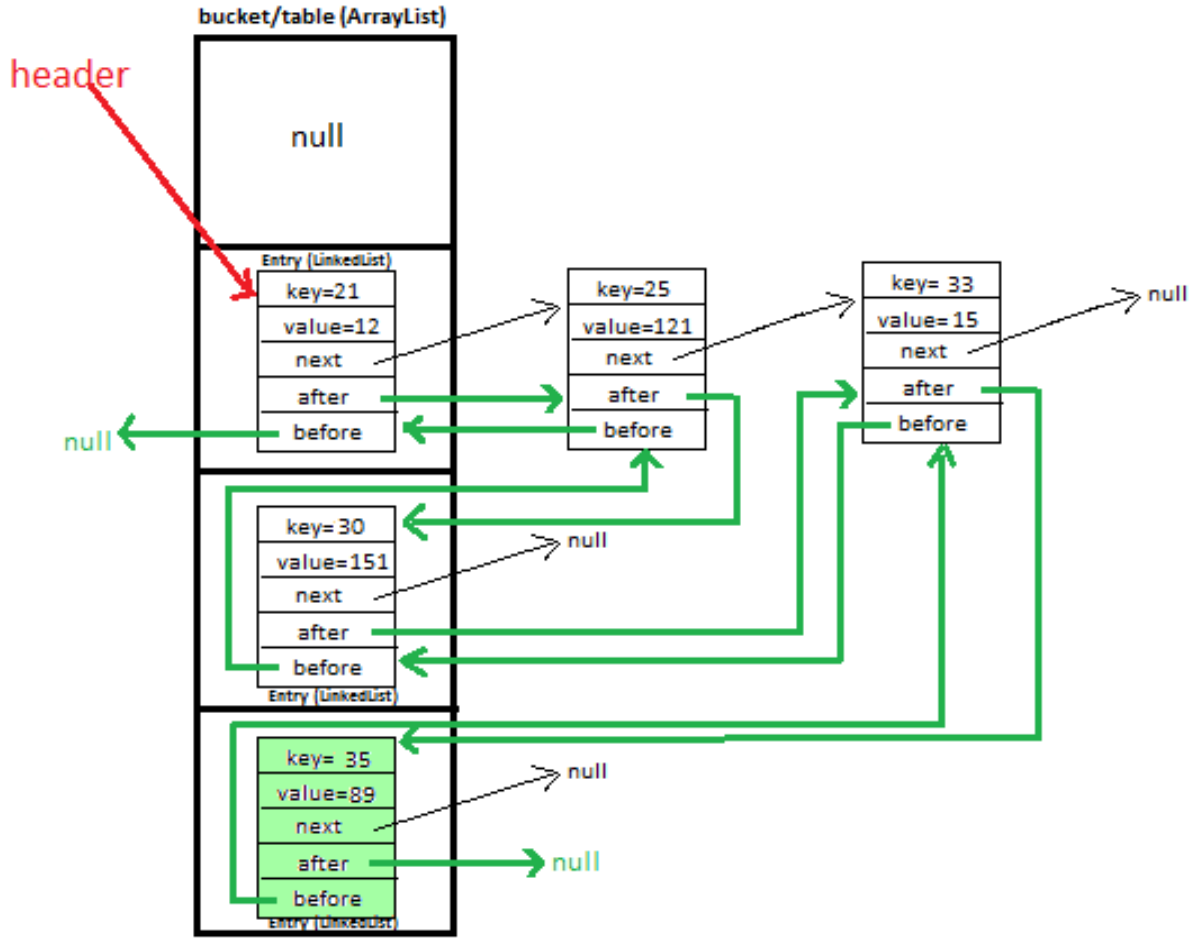
```
Iterator it = hashMap.entrySet().iterator();
while (it.hasNext()) {
        Map.Entry e = (Map.Entry)it.next();
        System.out.println(e.getKey() + " " + e.getValue().toString());
}
```

Map usage example

```
import java.util.*;

public class Agenda
{
   public static void main(String args[])
   {

//       HashMap global = new TreeMap();
         //Sort by String

//NO sort
Map<String,String> global = new
                    HashMap<String,String>();

        // Insertar valores "key"-"value" al HashMap
        global.put("Doctor", "(+52)-4000-5000");
        global.put("Casa", "(888)-4500-3400");
        global.put("Hermano", "(575)-2042-3233");
        global.put("Hermana", "(421)-1010-0020");
        global.put("Suegros", "(334)-6105-4334");
        global.put("Oficina", "(304)-5205-8454");
        global.put("Ana C.", "(756)-1205-3454");
        global.put("Luis G.", "(55)-9555-3270");
        global.put("Oficina 2", "(874)-2400-8600");

// Iterator to retrieve/print its values
for( Iterator it = global.keySet().iterator(); it.hasNext();) {
          String s = (String)it.next();
          String s1 = (String)global.get(s);
          System.out.println(s + " : " + s1);
}

// Define an array with some values
 String args1[] = {
         "Casa", "Ana C.", "Luis G."
 };

// Remove HaspMap item which are in the array HashMap con
en el array
for(int i = 0; i < args1.length; i++) {
         global.remove(args1[i]);
}


// Iterator to retrieve/print its values
for( Iterator it2 = global.keySet().iterator(); it2.hasNext();) {
          String s = (String)it2.next();
          String s1 = (String)global.get(s);
          System.out.println(s + " : " + s1);    }

   }
}
```

LinkedHashMap

- This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.
- This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).



Which Collection to use in which cases

- The following slides are from EDI (Data structures and information subject) and they show how to work with the different collections in an example. Depending on the collections, how the algorithm has to change

- The example is based on:
  - List of contact
  - Set of contact
  - Map of contact

- And it is shown how the operation add, duplicate and find and iterate the collections.

Exercise to do

- Evitar duplicados. En la sesión 9 se tendría el sig. código
  - Problema: Cada vez que se añade, se debe recorrer toda la lista

```
public boolean addContact(Contact c) {
  Iterator it=contactos.iterator();
  boolean enc=false;
  while (it.hasNext()){
        Contact aux= (Contact)it.next();
        if (aux.getPhoneNumber()==c.getPhoneNumber()){
           enc=true;
           break;
        }
  }
  if (!enc) return contactos.add(c); //Se añade sino esta
  return false;
}
```

Exercise to do

- Evitar duplicados. Sesión 10 → Usando un conjunto
  - **HashSet** controla automáticamente que no haya duplicado. Si el dato ya existe, no lo añade.

```
public class Hangout {

  private HashSet<Contact> contactos;

  public Hangout() {
        contactos = new HashSet<Contact>();
  }

  public boolean addContact(Contact c) {
        Return contactos.add(c);
  }
```

Exercise to do

- Evitar duplicados. Sesión 10 → Usando mapas
  - **HashMap:**
    - ✓ **No** controla automáticamente que no haya duplicado.
    - ✓ Pero su búsqueda interna es más eficiente que List
    - ✓ En este caso consideraremos la clave el teléfono del contacto pues en principio es única.

Clave: Integer -Teléfono
Valor/Objeto: Contact

```
public class Hangout {

  private HashMap<Integer,Contact> contactos;

  public Hangout() {
        contactos = new HashMap<Integer,Contact>();
  }

  public boolean addContact(Contact c) {
        if (!contactos.containsKey(c.getPhoneNumber()))
              return contactos.put(c.getPhoneNumber(), c);
        return false;

  }
```

¿Existe la clave?

Exercise to do

- Evitar duplicados.
  - List: Se tiene que recorrer toda la lista
  - Set: Controla él automáticamente que no haya repetido siempre que haya un hashcode() y equals()
  - Maps: Realiza una búsqueda eficiente para evitar que haya duplicado

Exercise to do

- Buscar un contacto. En la sesión 9 se tendría el sig. código
  - **Problema**: Se debe recorrer toda la lista

```
public void getContact(int phone) {
  Iterator it=contactos.iterator();
  boolean enc=false;
  while (it.hasNext()){
         Contact aux= (Contact)it.next();
         if (aux.getPhoneNumber()==phone){
            return aux;
         }
  }
  return null;
}
```

Exercise to do

- Evitar duplicados. Sesión 10 → Usando un conjunto
  - **Problema**: Se debe recorrer toda la lista. No aporta ventaja alguna sobre la lista

```
public void getContact(int phone) {
  Iterator it=contactos.iterator();
  boolean enc=false;
  while (it.hasNext()){
         Contact aux= (Contact)it.next();
         if (aux.getPhoneNumber()==phone){
            return aux;
         }
  }
  return null;
}
```

Exercise to do

- Evitar duplicados. Sesión 10 → Usando mapas
  - **HashMap:**
    - ✓ Su búsqueda es más eficiente que List

```
public Contact getContact(int phone) {
   return contactos.get(phone);

}
```

Exercise to do

- Obtener un objeto dado un valor.
  - List: Se tiene que recorrer toda la lista
  - Set: Se tiene que recorrer toda la lista
  - Maps: Tiene un método que permite eficientemente recuperar el contacto.
    - ¿Qué sucede si se desea buscar un contacto sin usar la clave? Por ejemplo a partir del nombre → En este caso se tiene que recorrer → Iterator

```
public Contact getContact(String name) {
  Iterator it = contactos.entrySet().iterator();
  while (it.hasNext()) {
        Map.Entry e = (Map.Entry)it.next();
        //Se usa e.getKey() obtener la clave
        //Se usa e.getValue() obtener el objeto
        Contact aux= (Contact)e.getValue();
        if (aux.getName().equals(name))
              return aux;
  }
  return null;
}
```

Choosing a Container

- We have seen a number of containers. We should know in what situations to use each of them
- We now when we need a List or a Set or a Map
- List is used when the order of elements are important
- Set is used when we don't care about order but we want to be sure there is no repeatition
- Map is used for making associations between keys and values (lookup tables)

Choosing between List implementations

	get	*add*	*contains*	*next*	*remove(O)*	*Iterator.remove*
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

```
--- Array as List ---
 size      get      set
   10      130      183
  100      130      164
 1000      129      165
10000      129      165
---------------------- ArrayList ----------------------
 size      add      get      set iteradd  insert  remove
   10      121      139      191     435    3952     446
  100       72      141      191     247    3934     296
 1000       98      141      194     839    2202     923
10000      122      144      190    6880   14042    7333
---------------------- LinkedList ----------------------
 size      add      get      set iteradd  insert  remove
   10      182      164      198     658     366     262
  100      106      202      230     457     108     201
 1000      133     1289     1353     430     136     239
10000      172    13648    13187     435     255     239
```

Choosing between Set implementations

	add	*contains*	*next*	*Note*
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

```
------------ TreeSet ------------
 size        add  contains   iterate
   10        746       173        89
  100        501       264        68
 1000        714       410        69
10000       1975       552        69
------------ HashSet ------------
 size        add  contains   iterate
   10        308        91        94
  100        178        75        73
 1000        216       110        72
10000        711       215       100
---------- LinkedHashSet ----------
 size        add  contains   iterate
   10        350        65        83
  100        270        74        55
 1000        303       111        54
10000       1615       256        58
```

Choosing between Maps

	get	*containsKey*	*next*	*Note*
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

Treemap

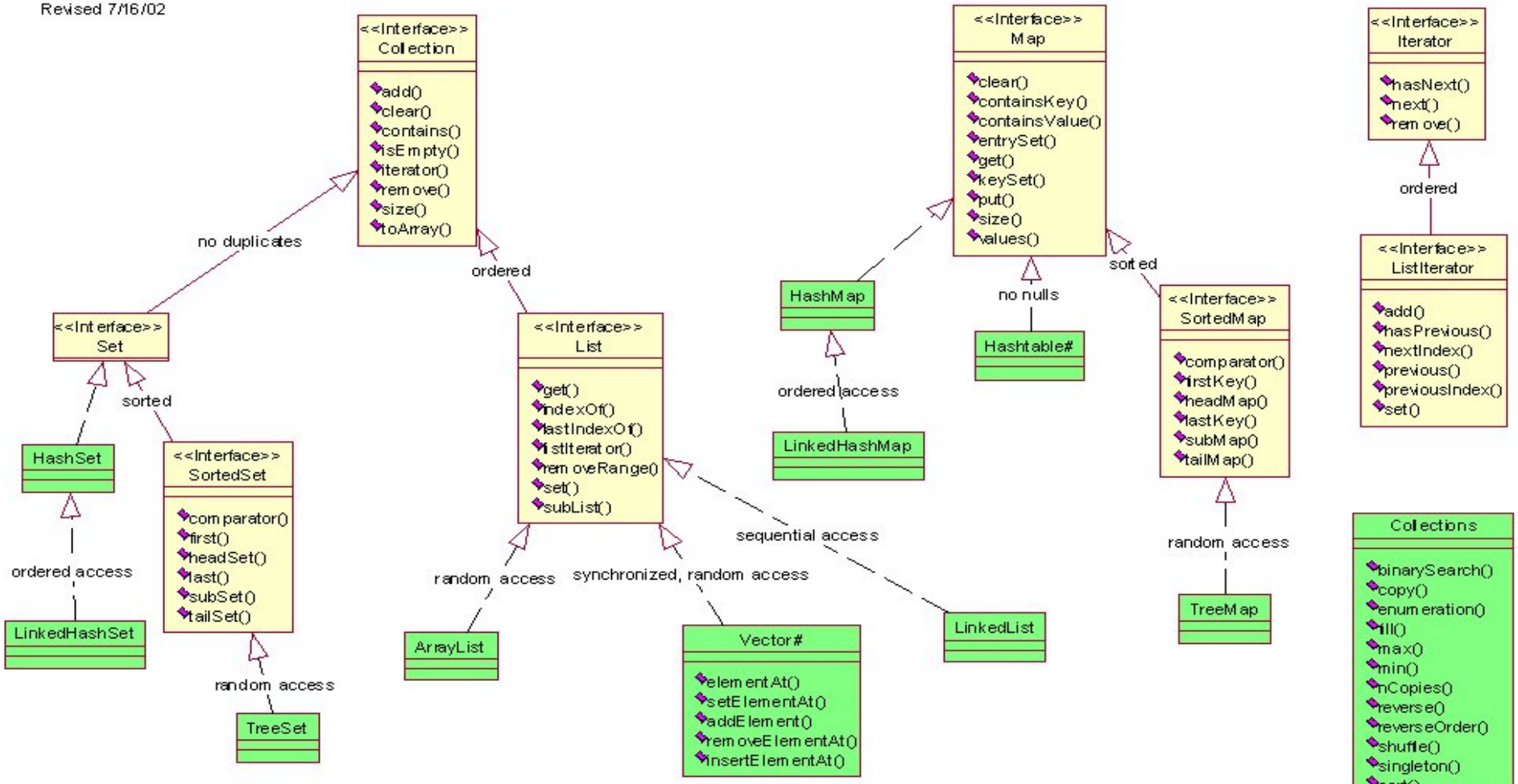
```
    10      748      168      100
   100      506      264       76
  1000      771      450       78
 10000     2962      561       83
---------- HashMap ----------
 size      put      get iterate
   10      281       76       93
  100      179       70       73
 1000      267      102       72
10000     1305      265       97
------- LinkedHashMap -------
 size      put      get iterate
   10      354      100       72
  100      273       89       50
 1000      385      222       56
10000     2787      341       56
------ IdentityHashMap ------
 size      put      get iterate
   10      290      144      101
  100      204      287      132
 1000      508      336       77
10000      767      266       56
-------- WeakHashMap --------
 size      put      get iterate
   10      484      146      151
  100      292      126      117
 1000      411      136      152
10000     2165      138      555
--------- Hashtable ---------
 size      put      get iterate
   10      264      113      113
  100      181      105       76
 1000      260      201       80
10000     1245      134       77
```

Difference among Maps and HashTable

Property	HashMap	TreeMap	LinkedHashMap	HashTable
Iteration Order	Random	Sorted according to natural **order of keys**	Sorted according to the **insertion order**.	Random
Efficiency: Get, Put, Remove, ContainsKey	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$
Null keys/values	allowed	Not-allowed*	allowed	Not-allowed
Interfaces	Map	Map, SortedMap, NavigableMap	Map	Map
Synchronized	Not instead use Collection.synchronizedMap(new HashMap())			Yes but prefer to use ConcurrentHashMap
Implementation	Buckets	Red-Black tree	HashTable and LinkedList using doubly linked list of buckets	Buckets
Comments	Efficient	Extra cost of maintaining TreeMap	Advantage of TreeMap without extra cost.	Obsolete

Java Collections Framework

Revised 7/16/02



Notes:

# denotes class is a legacy class.

Differences between Java and Java2:
* Non-legacy collections are not synchronized.
* Iterator modifications change the backing collection.

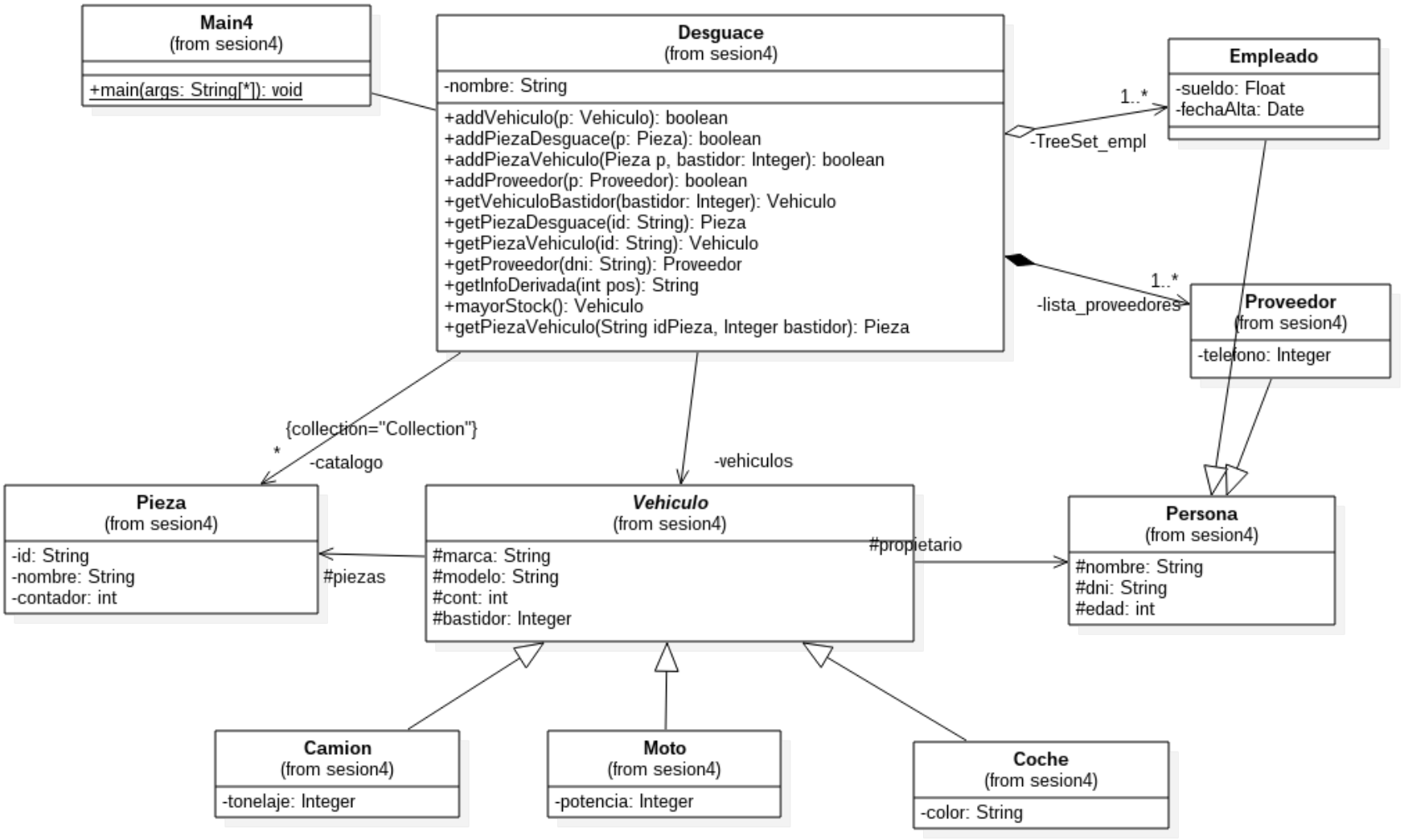
Not all classes, interfaces, or methods are shown.

- Añadir una clase Empleado (sueldo y fechaAlta) que herede de Persona así como un HashMap de **empleados** en Desguace.

- Debe realizarse una entrega en el campusvirtual basada en una batería de test de Desguace para los siguientes métodos (ver siguiente transparencia)

- Pruebas de la sesión 5
  - Constructor: Dejar el atributo tam en el CP aunque no se use
  - GetNombre y SetNombre
  - public Vehiculo getVehiculoBastidor(Integer bastidor);
  - public boolean addVehiculo(Vehiculo v);
  - public boolean addPiezaVehiculo(String id, Integer bastidor):
    - ✓ **Se recibe id en lugar de Pieza y se debe buscar en el catálogo**
  - public Vehiculo mayorStock() **usando Collections.sort (Vehículo implementa Comparable en base al número de piezas del Vehículo)**
  - public boolean addPiezaDesguace(Pieza p):
  - public Pieza getPiezaDesguace(String id):
  - public Pieza getPiezaVehiculo(String idP, Integer bastidor):
  - public boolean addEmpleado(Empleado e):
  - public boolean getEmpleado(String dni):
  - public float getMediaSueldo(): Calcula la media de los sueldos de los empleados

Diagrama de Clase



Bibliography

- Piensa en Java. 4ª Edición. Pearson Prentice Hall. ISBN 13: 9788489660342
- Core Java 2 Vol I. Fundamentos. Pearson Prentice Hall/Sun. ISBN 13: 9788420548326
- Core Java 2. Vol II. Características Avanzadas. Pearson Prentice Hall/Sun. ISBN 13: 9788483223109
- Programación, Algoritmos y ejecicios resultos en Java. Pearson Prentice Hall. ISBN 13: 9788420540245
- Estructuras de datos con Java. Diseño de estructuras y algoritmos. Pearson Addison Wesley. ISBN 13: 9788420550343