Ficha 10

Búsqueda por Dispersión: Hashing

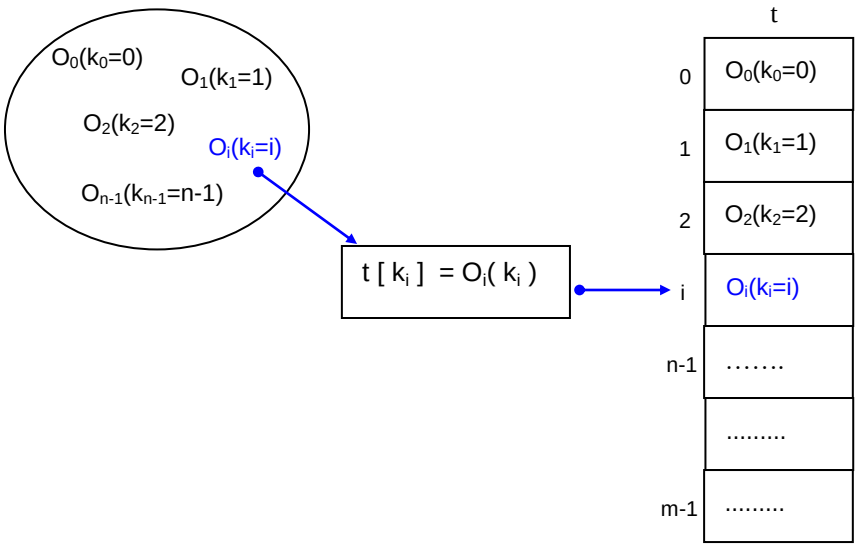
1.] Introducción y conceptos básicos.

Existen diversas estructuras de datos (y formas de organizarlas) que posibilitan *búsquedas* con diversos rendimientos en cuanto al tiempo para encontrar una clave en el peor caso o en el caso medio. Así, sabemos que si tenemos un *arreglo desordenado* o una *lista*, una búsqueda en el peor caso insumirá *O(n)* comparaciones. Si el *arreglo está ordenado* (y aprovechando su capacidad de acceso directo) una búsqueda insumirá *O(log(n))* aplicando el algoritmo de *búsqueda binaria*. Lo mismo ocurrirá (como veremos en una ficha posterior) si usamos un *árbol de búsqueda balanceado* (*AVL* o similar): las búsquedas insumirán aproximadamente un tiempo *O(log(n))*. Nos preguntamos si puede hacerse mejor. Es decir: ¿podemos diseñar una estructura de datos que permita organizar las claves u objetos contenidos en ella, de forma que el tiempo de búsqueda sea mejor que *O(log(n))*? Y si fuera posible, ¿qué tan mejor sería?

La respuesta es obvia y a la vez sorprendente: Se puede hacer, y el tiempo de búsqueda puede ser tan bueno como *O(1)* (o sea, *tiempo constante*: podemos buscar una clave de forma que el tiempo insumido sea constante, sin importar la cantidad de objetos que tenga la estructura). La forma más obvia de lograr tal meta, consiste en dimensionar una *tabla* (o sea, un *arreglo*) suficientemente grande para que entren todos los objetos que se necesitan almacenar, y luego grabar cada objeto en la casilla cuyo índice coincida con el valor usado como *clave* del objeto (normalmente, en un ambiente polimórfico se usa como clave el valor retornado por el método *hashCode()* de cada objeto, que se hereda desde *Object* y que cada programador redefine de manera sencilla).

Sea (por ejemplo) un conjunto de n objetos $\{O_0, O_2, O_3, ... O_{n-1}\}$, tales que k_i es el valor *hashCode()* del objeto O_i. Si t es una tabla de tamaño m con $m >= n$, entonces $t[k_i]$ debe usarse para almacenar el objeto $O_i(k_i)$ (ver figura en la página siguiente).

De esta forma, la recuperación de un objeto es directa: sólo necesitamos saber su clave (su *hashCode()*) y ella nos dice en qué casilla buscar. Sin embargo, sabemos que esto no siempre es viable: si la clave puede ser un número cualquiera en cualquier rango numérico, entonces la tabla debería ser extremadamente grande para posibilitar que cualquier valor sea un índice válido. Un ejemplo conocido se da cuando la clave es el número de *dni* de cualquier ciudadano argentino: como ese número tiene un máximo de 8 dígitos, la tabla debería ser tan grande como de *100 millones de casillas* para que cualquier *dni* pueda ser un índice válido... aún cuando luego sólo se almacenen unas pocas docenas de objetos.



La técnica conocida como *dispersión de claves* o simplemente *hashing*, busca resolver este problema usando una función designada como *función de dispersión*, *función hash* o *función de transformación*. Esta función es provista por el programador y la idea es que tome como parámetro una *clave de un objeto* (el valor *hashCode()* de ese objeto...) y retorne un índice válido para una tabla con capacidad para *m* objetos (de aquí en adelante, diremos que esa tabla tiene tamaño *m*). La manera obvia de implementar una función *h* de dispersión para una tabla de tamaño *m*, es hacer que *h* retorne el resto de dividir la clave del objeto por el tamaño *m* de la tabla. Si la clase *TablaHash* se implementa en memoria con un arreglo de objetos, podría verse así (en forma muy simplificada):

```
public class TablaHash<>
{
      private Object  t[];    // la tabla, como un arreglo
      ...

      public int h (Òbject obj)
      {
            int m = t.length;
            int k = obj.hashCode();
            return k % m;
      }
}
```

Note que el método *hashCode()* disponible en cada objeto, permite obtener un número entero que puede servir como identificador numérico para ese objeto, aún en situaciones en que el objeto pudiera disponer (a modo de clave) de atributos que no sean números enteros: el programador dispone qué es lo que debe devolver *hashCode()*, y si debe obtener un número a partir de un *String* (por ejemplo), puede hacer que se retorne el *hashCode()* de ese *String*. La siguiente clase muestra la forma:

```
public class Automovil implements Comparable
{
      private String patente;
      ...

      public int hashCode()
```

```
        {
              return patente.hashCode();
        }
}
```

Como sea, el método *hashCode()* debería retornar un número entero que en la medida de lo posible sea diferente para dos objetos que sean distintos según el método *equals()*. Sin embargo, esto no es obligatorio: dos cadenas distintas podrían generar el mismo *hashCode()*, o dos objetos de cualquier clase del programador podrían tener el mismo *hashCode()* si así lo dispuso el programador. En cualquier caso, si dos objetos diferentes retornan valores distintos de *hashCode()* se disminuye el riesgo de una *colisión*. Dicho sea de paso, la convención que siguen los programadores (a sugerencia de la documentación de Java), es que si se decide redefinir el método *hashCode()* en una clase, se redefina también *equals()*, de forma que se cumpla la siguiente idea: si *equals()* indica que dos objetos son iguales, entonces el valor retornado por *hashCode()* para esos dos objetos debería ser el mismo. En los ejemplos que acompañan a esta ficha, mostramos algunas formas de cumplir estas convenciones.

2.] Colisión de claves. Estrategias de resolución de colisiones.

Si se usa la técnica de dispersión de claves o hashing, podrían aparecer problemas: una *colisión* es una situación en la que *dos objetos diferentes obtienen el mismo índice para entrar en la tabla*. Es decir, la función de dispersión calcula el *mismo* valor para los valores de código *hash* de esos dos objetos. Esto puede producirse porque ambos objetos tienen la misma clave o atributos a partir de los cuales se basa el cálculo de código hash en el método *hashCode()*, o porque teniendo claves diferentes la función de dispersión calcula el mismo resto contra el tamaño de la tabla (las claves o valores de hashCode() son *congruentes* al tamaño de la tabla):

```
obj1.equals(obj2) == false  pero  h(obj1)== h(obj2)
```

Como no es posible almacenar dos o más objetos en la misma posición de una tabla, se usan estrategias para evitar este problema, conocidas como *técnicas de resolución de colisiones*. Existen dos de estas técnicas que son consideradas las más tradicionales:

a. Resolución de colisiones por direccionamiento abierto:

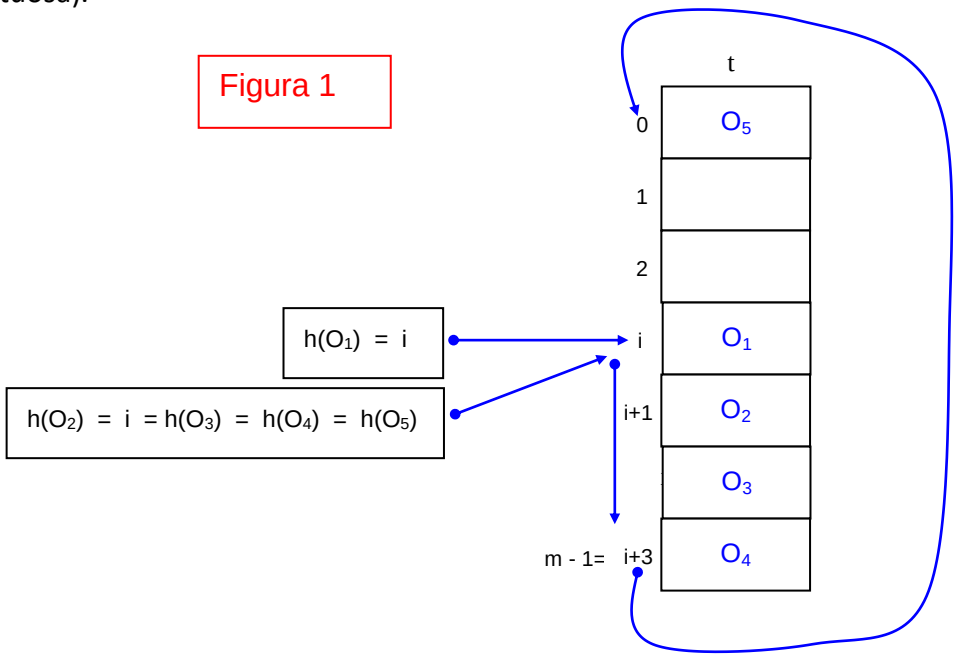
Cada entrada de la tabla almacena un objeto (y sólo uno). Al principio, cada casilla está vacía o *abierta* (de allí el nombre de esta técnica). Si un objeto O_1 pide entrar en una casilla *i* y la misma está abierta, O_1 se almacena en ella. El índice *i* de esa casilla se dice la *dirección madre de* O_1. A partir de este momento, la casilla *i* está cerrada.

Si otro objeto O_2 pide entrar en la tabla y colisiona en la casilla *i*, se prueba en la casilla *i+1*. Si está abierta, O_2 se almacena en ella. Pero si está *cerrada*, se sigue probando con *i+2*, *i+3*, *i+4*... (haciendo una *exploración lineal*) hasta llegar a una casilla abierta y en ella se almacena el nuevo objeto. Notar que entonces, varios objetos serán almacenados fuera de su dirección madre (aunque no lejos de ella en sentido secuencial). Si en la

exploración lineal se llega hasta el final de la tabla sin encontrar una casilla abierta, se sigue desde la casilla cero (circularizando la tabla) (ver figura en página siguiente).

Para buscar un objeto, se obtiene su dirección madre con la función *h()*, y se entra en la tabla en esa dirección. Si el objeto en esa casilla no es el buscado, se explora hacia abajo hasta encontrarlo (búsqueda exitosa), o hasta encontrar una casilla abierta (búsqueda infructuosa).

Figura 1



El problema es que en algún momento se querrá eliminar un objeto de la tabla (o sea, hacer una *baja*). En principio, usamos marcado lógico para eliminar el objeto. Para ello, se debe buscar el objeto con el procedimiento anterior y una vez hallado *marcar* la casilla como *vacía* o *abierta*. Pero eso invalidaría el procedimiento de búsqueda: si este debe terminar al encontrar una casilla abierta, la búsqueda podría darse por infructuosa cuando quizás el objeto esté más abajo en la tabla. Suponga que en la gráfica anterior se elimina el objeto O_3 quedando abierta su casilla. Si ahora se busca el objeto O_4 o el objeto O_5 la búsqueda fallará... (ver figura en la página siguiente).

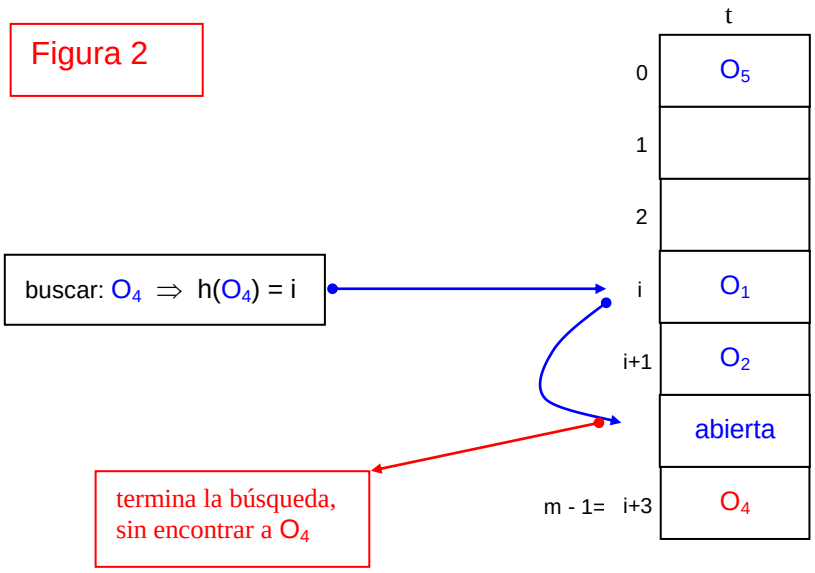
La solución consiste en que cada casilla se marque en uno de tres estados posibles:

- ***abierta***: la casilla está vacía, y nunca hubo en ella un objeto.
- ***cerrada***: la casilla está ocupada por un objeto que no fue borrado.
- ***tumba***: la casilla está vacía, pero porque se borró un objeto de ella[1].

Y de esta forma, la búsqueda puede plantearse así: obtenga la dirección madre del objeto a buscar. Entre en esa casilla. Si no contiene al objeto, realice una exploración lineal hacia

[1] Puede parecer algo tenebrosa la designación de *tumba* para una casilla, pero se han seguido aquí terminologías que son clásicas en diversos autores. Es común marcar a estas casillas con el valor *tombstone* (lápida), tal que *tombstone* es alguna constante declarada en alguna clase.

abajo y en forma circular si fuera necesario, hasta dar con el objeto o enctontrar una casilla *abierta* (*no una casilla cerrada o una tumba*).



Tenga cuidado al hacer una inserción: en general en una tabla hash no se admiten valores repetidos, por lo que antes de insertar un objeto debe buscar el mismo en la tabla. Si no se encuentra, se inserta. Debe resistir la tentación de insertar el objeto en la primera tumba que encuentre, antes de terminar la búsqueda, pues el objeto podría estar más abajo... Lo que sí puede hacer es guardar el índice de la primera casilla tipo tumba que encuentre, y si luego descubre que el objeto no está en la tabla, guardarlo en ella.

En general la técnica de resolución de colisiones por direccionamiento abierto permite encontrar un objeto en tiempo constante: puede verse que se requiere un acceso directo a la dirección madre, y una corta exploración lineal en el peor caso para hallar el objeto, pero si la tabla está bien implementada esos recorridos lineales tendrán una longitud que depende de n (la cantidad de objetos de la tabla).

Puede preverse que a medida que la tabla se llene, las exploraciones lineales serán cada vez más largas, y para evitar que efectivamente la longitud de cada tramo a recorrer se se parezca a *n*, entonces la tabla debería empezar con un tamaño mayor al número de objetos esperados, y controlar el *porcentaje de ocupación*. Si el mismo llega a cierto valor crítico (por ejemplo, el 50%), se debería *redimensionar* la tabla (analizaremos brevemente este proceso al final de esta ficha de clase).

Por otra parte, aún cuando el porcentaje de ocupación de la tabla sea adecuado, el direccionamiento abierto tiende a presentar comportamientos no deseados conocidos como *agrupamiento primario* y *agrupamiento secundario*. El *agrupamiento primario* es la *tendencia de los objetos de la tabla a formar islas o grupos secuenciales dentro de esa tabla*. La tabla presenta muchos espacios libres, pero los objetos tienden a caer cerca de las mismas direcciones madre. Esto provoca que para ciertas claves, la inserción

comience a ser costosa pues debe acomodarse dentro de un grupo ya grande, y además la isla crece en tamaño con la inserción de la nueva clave. Una explicación para este comportamiento es que muchas veces los objetos vienen en secuencias de entrada no aleatorias, haciendo que la función de dispersión acomode juntos a varios de esos objetos.

Una solución para el agrupamiento primario consiste en realizar lo que se conoce como *exploración cuadrática* (en lugar de exploración lineal): Si la casilla *i* está ocupada, se sigue con *i + 1* y luego con *i + 4*, *i + 9*, ... $i + j^2$ (con *j = 1, 2, 3, ...*) Está claro que el agupamiento primario se rompe con este tipo de exploración, pero ahora se produce el *agrupamiento secundario*: dada una dirección madre *i*, siempre se exploran las mismas casillas de allí en adelante. Esto lleva a otro posible problema: la exploración cuadrática *podría no garantizar que una clave se inserte finalmente en la tabla*, aún habiendo lugar libre... (intente el estudiante mostrar cómo podría pasar ésto, haciendo un gráfico de una tabla pequeña).

No obstante, se puede demostrar (aunque no lo haremos aquí) que *si el tamaño de la tabla es un número primo* y el *porcentaje de ocupación no es mayor al 50% de la tabla*, entonces la exploración cuadrática *garantiza* que la clave será insertada.

En la práctica, ***no hay nada*** que justifique que se deje de lado la exploración cuadrática a favor de la lineal. El sólo hecho de evitar el agrupamiento primario es suficiente ventaja. En la implementación, el programador debe asegurarse que el tamaño de la tabla sea un número primo, y que la tabla se mantega siempre medio vacía. Como ayuda, va el siguiente método que facilita encontrar el siguiente primo mayor que un número *n* dado[2]:

```
private static final int siguientePrimo ( int  n )
{
      if ( n % 2  == 0)    n++;
      for ( ; !esPrimo(n); n+=2 ) ;
      return n;
}
```

En esta rutina, el método *boolean esPrimo(int n)* debe verificar los divisores impares posibles de *n* (el algoritmo mostrado garantiza que luego del *if*, *n* no es par) entre 3 y raiz de n. Si ningún número en ese intervalo divide a *n*, entonces *n* es primo. Se deja para el estudiante...

Observemos que el tamaño de la tabla siempre debería ser un número primo, y esto debe garantizarse incluso si la clase implementada brinda un constructor que acepte como parámetro el tamaño deseado para la tabla a crear. Sin importar el valor de ese parámetro, el constructor deberá crear una tabla de tamaño *m* primo, tal que *m* sea mayor que el parámetro enviado.

b. Resolución de colisiones por listas de desborde:

[2] Tomado de "*Estructuras de datos en Java*" de Mark Allen Weiss – Editorial Addison Wesley.

Cada entrada de la tabla almacena *una lista*, en la que se guardan los objetos que colisionaron en esa entrada (vea figura más abajo). Al insertar un objeto, la función de dispersión indica en qué lista se debe agregar el objeto. De nuevo, esto implica que para buscar un objeto se debe hacer un acceso directo y un corto recorrido sencuencial en una lista, manteniendo el tiempo de búsqueda en orden constante: en una implementación bien hecha, el tamaño de cad lista no depende de *n*..

Está claro que ahora no es importante el porcentaje de carga de la tabla, sino la *longitud promedio de las listas* en ella, pues ese promedio indica el rendimiento promedio de una búsqueda. Una tabla de tamaño inicial adecuado para asegurarse que cada lista tenga entre tres y diez nodos (sea cual sea *n*) funcionará bien, lo cual permite pensar en tablas con un tamaño *m* no tan grande como para tener un amplio espacio usado en listas que podrían estar vacias...

Si se puede asegurar que las listas no crecerán demasiado, entonces puede evitarse tener que ampliar el tamaño de la tabla y volver a dispersar todas las claves en ella. Por este motivo, y su sencillez de implementación si se tiene ya una clase desarrollada para gestionar listas, esta técnica suele ser la preferida por muchos programadores, sin contar el hecho que en algunos lenguajes es una opción casi obligada pues esos lenguajes no cuentan con la posibilidad de expandir un arreglo en forma dinámica. En el modelo *TSBHashtable* que acompaña a esta ficha, se implementa esta idea de resolución de colisiones en la clase *TSBHashtable*.

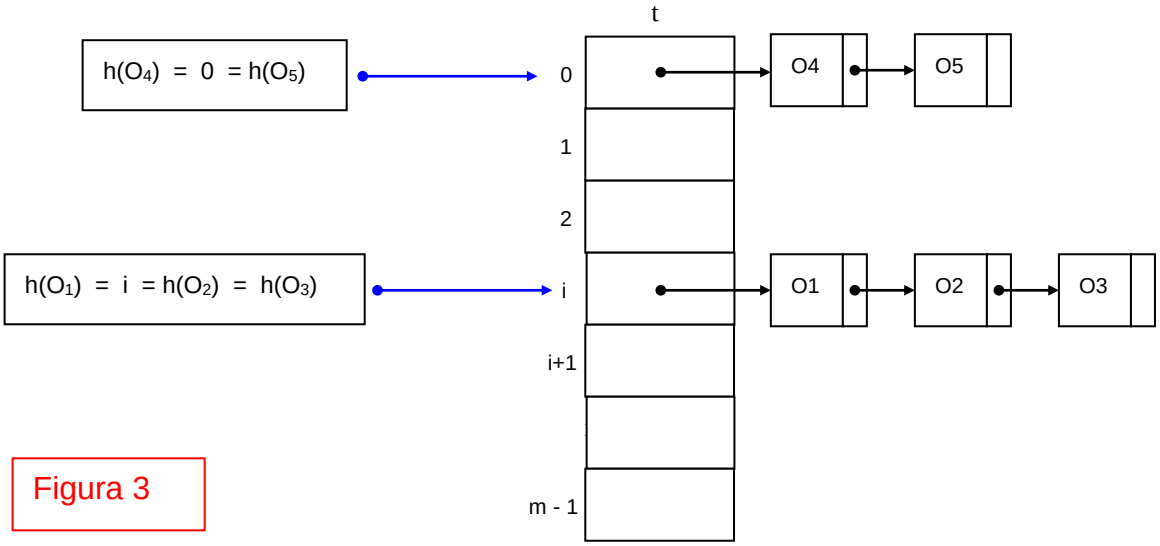


Figura 3

3.] Control de carga y aumento del tamaño de la tabla.

Un control importante que debe realizarse es el del *porcentaje de carga de la tabla* (si se usa direccionamiento abierto) o el *tamaño promedio de las listas de desborde* (si se usa esa técnica). En ambos casos, se debería monitorear la carga de la tabla o el largo promedio de las listas, y en caso de llegar al punto de intervención, aumentar el tamaño de la tabla (por ejemplo, en un 50% o hasta el número primo siguiente a ese 50%, o al doble del tamaño anterior, por citar algunas estrategias).

El proceso es simple pero debe hacerse con cuidado: se crea la nueva tabla, se toman todos los objetos que estaban en la anterior, y se *redispersan* en la nueva tabla: no se puede sólo *copiar* los registros de una a otra, pues los tamaños de ambas no coinciden y los valores calculados por *h()* no serán consistentes. Al terminar, la vieja tabla se elimina, y la nueva debe tomar el nombre de la eliminada (o ser apuntada por la referencia que antes apuntaba a la vieja tabla). El proceso completo, si la tabla original tenía *n* objetos y recordando que cada nueva inserción se hace en tiempo constante, llevará entonces un tiempo *O(n)*: *n* reinserciones a un costo de *O(1)* cada una…

En la clase *TSBHashtable* del modelo que acompaña a esta ficha, se incluye el método *rehash()* para redispersar la tabla en otra mayor. Se ha reusado la clase *TSBArrayList* para almacenar objetos en listas de esa clase cuando haya una colisión. La clase *TSBHashtable* dispone de un método privado *averageLength()* que calcula la longitud promedio de las listas en la tabla. Cada vez que se invoca al método de inserción en la tabla (el método *put()*), se controla si esa longitud es mayor a cierto valor crítico (que se pasa como parámetro al constructor y se almacena convenientemente en un atributo de la clase), y en ese caso se invoca al método *rehash()*.

Tanto se si aplica direccionamiento abierto como si se trabaja con listas de desborde, el valor crítico a partir de cual se decide lanzar un proceso de *rehash()* se suele designar con el nombre de *factor de carga de la tabla*, y dependiendo del tipo de implementación tendrá una interpretación diferente.

4.] Detalles de implementación: Hashing con listas de desborde.

En el modelo *TSBHastable* que acompaña a esta ficha, hemos implementado en forma rigurosa la estrategia de tabla hash con listas de desborde para resolver el problema de las colisiones. Como dijimos, las listas que se almacenan en la tabla son a su vez, objetos de la clase *TSBArrayList* (que ya hemos implementado anteriormente).

Para continuar con la idea de emular la funcionalidad de clases ya provistas por Java, podemos decir que el lenguaje Java provee al menos tres clases que implementan en forma nativa la idea de tabla hash: una es la clase *java.util.HashSet*, otra es *java.util.HashMap*, y la tercera es *java.util.Hashtable*. Las tres usan listas de desborde para resolver colisiones.

La clase *java.util.HashSet* permite almacenar objetos usando como clave o identificador numérico directamente al valor retornado por *hashCode()* para esos objetos. Cada objeto que se desee almacenar en la tabla, debe proveer una implementación adecuada de ese método, y la propia clase *HashSet* invoca al método del objeto a insertar, calculando así el índice de la casilla donde será almacenado.

Las clase *java.util.HashMap* y *java.util.Hashtable* por su parte, están ambas pensadas para almacenar *pares de objetos* de la forma *(key, value)*, en donde *key* es un objeto de cualquier clase cuyo *hashCode()* se usa para calcular la posición de la tabla donde se almacenará el par completo, y *value* es otro objeto que en la tabla queda asociado al objeto *key* (y almacenado con él). Una estructura de datos que aplica esta idea, se suele designar con el nombre

genérico de *Map* (debido a que permite el *mapeo* de un conjunto de claves sobre un conjunto de valores). Si el método *put()* es el que permite agregar un par en la tabla, una invocación de la forma *put(1, "Argentina")* agregará en la tabla un *par* de objetos: uno de tipo *Integer* valiendo 1, y el otro de tipo *String* valiendo "Argentina". El objeto *Integer* cuyo valor es 1 es la *clave* (key) con que se identifica dentro de la tabla al objeto *String* con valor "Argentina" (el *value* del par).

Las dos clases *java.util.HashMap* y *java.util.Hashtable* son casi en todo equivalentes, salvo por dos detalles: En primer lugar, la clase *java.util.Hashtable* no admite pares en los que alguno de sus componentes sea *null* (ni los objetos *key* ni los *value* pueden ser *null*), mientras que la clase *java.util.HashMap* sí los admite. Y en segundo lugar, la clase *java.util.Hashtable* es *thread-safe* (algo como: *a salvo de amenazas* en un hilo de ejecución): el estado de una instancia de esta clase siempre está bien definido incluso en un ambiente de múltiples hilos concurrentes accediendo y modificando a ese objeto, mientras que eso no vale para la clase *java.utilHashMap*. Si se espera usar una tabla hash en un ambiente multihilos, debería usarse una *Hashtable*, mientras que una *HashMap* puede usarse sin problemas en un ambiente no concurrente.

En nuestro modelo hemos seguido un esquema mixto: nuestra clase *TSBHashtable* emula en forma tan detallada como ha sido posible el funcionamiento de la clase *java.util.HashMap* (en el sentido de no ser *thread-safe*), pero sin permitir valores *null* (como hace la clase *java.util.Hashtable*). Sólo por razones de similitud conceptual con el tema que se está estudiando, el nombre de nuestra clase es *TBSHahstable* (y no *TSBHashMap*), aunque dado el esquema mixto que hemos aplicado, cualquiera de los dos nombres sería igualmente bueno.

Pero hacemos (nobleza obliga) una aclaración: en este caso no hemos aplicado la estrategia *clean room* para intentar replicar todo el diseño, debido a que las clases *java.util.Hashtable* y java.util.HashMap utilizan y aplican mecanismos internos, interfaces, subinterfaces, iteradores y vistas que las hacen particularmente complejas. Nuestra implementación no es exactamente idéntica a ninguna de las dos (como veremos) pero nos hemos basado en algunos detalles de implementación del código fuente de las clases nativas para terminar de cerrar algunos aspectos y manternos fieles a la idea nativa.

Por lo pronto, la clase *java.util.Hashtable* deriva de la clase *java.util.Dictionary*, originalmente sugerida por la documentación oficial para ser la base de toda estructura de datos que represente un mapeo de claves sobre un conjunto de valores. Sin embargo, en versiones más recientes de Java esta clase se ha marcado como obsoleta, y la nueva convención y recomendación es directamente implementar la interface *java.util.Map* (que provee todas la definiciones de métodos que pueden ser requeridas para implemenar una estructura con mapeo de claves sobre valores). Y efectivamente eso es lo que hacemos con nuestra clase *TSBHashtable*:

```
public class TSBHashtable<K, V> implements Map<K, V>, Cloneable, Serializable
```

La clase está parametrizada con el mecanismo generics para aceptar dos tipos de objetos: los objetos de tipo *K* serán los que se usen como *claves* (*keys*) en el map, y los objetos de la clase *V* serán los que se usen como *valores* (*values*) asociados a las claves. La clase implementa la inteface *Map<K , V>* (también parametrizada) y las ya conocidas *Cloneable* y *Serializable*. En cuanto a los atributos, la clase incluye los siguientes:

```
public class TSBHashtable<K,V> implements Map<K,V>, Cloneable, Serializable
{
        // el tamaño máximo que podrá tener el arreglo de soporte...
        private final static int MAX_SIZE = Integer.MAX_VALUE;

        // la tabla hash: el arreglo que contiene las listas de desborde...
        private TSBArrayList<Map.Entry<K, V>> table[];

        // el tamaño inicial de la tabla (tamaño con el que fue creada)...
        private int initial_capacity;

        // la cantidad de objetos que contiene la tabla en TODAS sus listas...
        private int count;

        // el factor de carga para calcular si hace falta un rehashing...
        private float load_factor;


        //********************** Atributos para gestionar vistas.
        private transient Set<K> keySet = null;
        private transient Set<Map.Entry<K,V>> entrySet = null;
        private transient Collection<V> values = null;

        // conteo de operaciones de cambio de tamaño (fail-fast iterator).
        protected transient int modCount;

       // resto de la clase aquí…
}
```

Como esencialmente se trata de crear un arreglo, y ese arreglo luego irá creciendo a medida que sea necesario, la constante *MAX_SIZE* se usa para almacenar (y luego controlar) el máximo tamaño que podrá tener ese arreglo (en este caso, el máximo valor que puede tomar una variable de tipo *int* en Java).

El atributo *table* es la tabla propiamente dicha. En ella se almacenan las listas de desborde, que son instancias de la clase *TSBArrayList*. En cada una de esas listas, se deben almacenar los pares de objetos *(key, value)*, pero si se está trabajando con la interface *Map*, Java espera que la clase que represente a esos pares de objetos implemente la interface específica *Entry* definida dentro de la interface *Map* (de hecho, se llama *Map.Entry*): Esta interface es *pública*, *estática* e *interna* de la interface *Map* (por eso el operador punto para entrar a ella desde *Map*, y por eso se puede usar el nombre de la interface *Map* para acceder a *Entry*). El programador debe definir su propia clase para representar pares de objetos, y hacer que esa clase implemente *Map.Entry*. En nuestro caso, hemos definido dentro de *TSBHashtable* una clase interna llamada *Entry* para cumplir con ese requisito:

```
public class TSBHashtable<K,V> implements Map<K,V>, Cloneable, Serializable
{
        // el tamaño máximo que podrá tener el arreglo de soporte...
        private final static int MAX_SIZE = Integer.MAX_VALUE;

        // la tabla hash: el arreglo que contiene las listas de desborde...
        private TSBArrayList<Map.Entry<K, V>> table[];

       // resto de los atributos y métodos aquí…

        // clase interna para representar los pares de objetos…
        private class Entry<K, V> implements Map.Entry<K, V>
        {
           private K key;
           private V value;

           public Entry(K key, V value)
           {
              if(key == null || value == null)
              {
                 throw new IllegalArgumentException("Entry(): parámetro null...");
              }
              this.key = key;
              this.value = value;
           }

           @Override
           public K getKey()
           {
              return key;
           }

           @Override
           public V getValue()
           {
              return value;
           }

           @Override
           public V setValue(V value)
           {
              if(value == null)
              {
                 throw new IllegalArgumentException("setValue(): parámetro null...");
              }

              V old = this.value;
              this.value = value;
              return old;
           }

           @Override
           public int hashCode()
           {
```

```
            int hash = 7;
            hash = 61 * hash + Objects.hashCode(this.key);
            hash = 61 * hash + Objects.hashCode(this.value);
            return hash;
        }

        @Override
        public boolean equals(Object obj)
        {
            if (this == obj) { return true; }
            if (obj == null) { return false; }
            if (this.getClass() != obj.getClass()) { return false; }

            final Entry other = (Entry) obj;
            if (!Objects.equals(this.key, other.key)) { return false; }
            if (!Objects.equals(this.value, other.value)) { return false; }
            return true;
        }

        @Override
        public String toString()
        {
            return "(" + key.toString() + ", " + value.toString() + ")";
        }
    }
    // resto de la clase aquí…
}
```

La clase *Entry* dispone de los dos atributos esperables: la clave *key de tipo K*, y el *valor value de tipo V*. La interface *Map.Entry* sólo obliga a incluir los métodos *getKey()*, *getValue()* y *setValue()*, pero es prácticamente mandatorio también incluir en ella redefiniciones correctas de *equals()*, *hashCode()* y *toString()*. Note que el único constructor que hemos incluido en esta sencilla clase, no admite que los valores finales a almacenar en *key* o en *value* sean *null*, y tampoco lo permite el método *setValue()* (no hay un método *setKey()*). Lo que esto implica es simple: la clase *TSBHashtable* no admitirá pares *(key, value)* en los que alguno de los objetos sea *null* (a diferencia de la clase *java.util.HashMap*, que como vimos, sí lo permite).

La forma en que está programado el método *hashCode()* de la clase *Entry* sirve como modelo y ayuda para fijar ideas: sabemos que el valor retornado por el método *hashCode()* de un objeto es el que debe ser usado luego para calcular en qué casilla de una tabla será almacenado ese objeto, y que el truco básico para hacer ese cálculo es dividir el valor provisto por *hashCode()* por el tamaño de la tabla y tomar el resto de esa división. El problema es que si los valores retornados por *hashCode()* responden a algún tipo de secuencia predecible o que siga algún patrón, entonces puede producirse un elevado número de colisiones. Por ejemplo, si el *hashCode()* a retornar por los objetos de una clase *Empleado* fuese simplemente el número de legajo de esos empleados, y en un momento dado se pidiese procesar los datos sólo de los empleados cuyo legajo termine en 0 o en 5 o en 0, entonces una tabla cuyo tamaño sea 5 provocaría que todos los registros fuesen

mapeados a las listas de las casillas 0 o 5, dejando las otras vacías y forzando ineficientes situaciones de *rehash* sólo por el desborde de dos de sus listas…

Para ayudar a evitar ese tipo de problemas, el método *hashCode()* de una clase debería *expandir* el valor a partir del cual comienza el cálculo, sumando y/o multiplicando a ese valor por números que en la medida de lo posible sean primos (note que eso es lo que se hizo en el método *hashCode()* de la calse *Entry*). Esto disminuye notablemente la probabilidad de que al dividir por cierto número fijo se obtengan restos iguales, sobre todo si ese número fijo es alto.

El atrubuto *initial_capacity* de la clase *TSBHashtable* almacena simplemente el valor del tamaño con el que la tabla fue creada la primera vez, para poder volver a ese tamaño en ocasión de invocar al método *clear()* (que vacía la tabla y la reconstituye a su tamaño original). El atributo *count* se usa para llevar la cuenta de la cantidad total de pares que contiene la tabla entre todas sus listas.

El atributo *load_factor* es un valor en coma flotante que se usa para representar el *factor de carga de la tabla*, usado para determinar si es necsario o no un proceso de *rehash*. En la clase *java.util.Hashtable*, un factor de carga (por ejemplo) de 0.75 indica que se debe hacer un *rehash* cuando la cantidad de casillas ocupadas en la tabla llegue al 75% del tamaño de esa tabla. En nuestro caso, no estamos usando el concepto de "porcentaje de ocupación" sino el de "longitud promedio de las listas de desborde", por lo que el valor de *load_factor* debería ser un número mayor a 1 y entero. Sin embargo, para mantener el espíritu de la clase original, hemos preservado la idea de que el factor de carga sea un número flotante mayor a 0 y menor que 1, con la siguiente convención: internamente, ese valor será multiplicado por 10 y el resultado será la longitud promedio que use la tabla para activar o no un rehash. Así, si *load_factor = 0.8*, entonces se lanzará un *rehash* cuando la longitud promedio de todas las listas sea mayor o igual a 0.8 * 10 = 8 elementos por lista.

El atributo *modCount* se usa para activar el ya conocido mecanismo *fail-fast iterator*, pero ahora de forma manual. Cuando hicimos la implementación de la clase *TSBArrayList*, ese atributo venía heredado desde la clase *AbstractList*, de forma que el mecanismo de control *fail fast* venía ya implementado: el programador sólo tenía que sumar 1 a ese atributo cada vez que se activase una operación que modifique el tamaño de la lista. En el caso de la clase *TSBHashtable*, ese atributo no viene heredado desde ninguna superclase y el mecanismo debe ser implementado manualmente. Como veremos, eso no es complejo de hacer.

Finalmente quedan tres atributos especiales llamados *keySet*, *entrySet* y *values*. La clase *TSBHashtable* (y en general, cualquier clase que represente un *Map*, como *java.util.Hashtable* o *java.util.HashMap*) no provee iteradores. Ninguna de estas clases implementa la interface *Iterable*, por lo que no se dispone de un método que pueda crear un iterador. La idea es que un *Map* es una estructura de datos internamente desordenada y en la que (en principio) no es posible predecir ni fijar en qué orden están ubicados sus elementos, y eso hace que la creación de un iterador propiamente dicho no forme parte de las responsabilidades de la clase. Sin embargo, en distintas situaciones de un modo u otro se

querrá contar con una forma de poder acceder en forma secuencial a todos y cada uno de los pares contenidos en el *Map*. La solución prevista por Java es crear *vistas* del contenido del *Map*, y los tres atributos citados (*keySet*, *entrySet* y *values*) son atributos para referenciar y gestionar esas *vistas*.

En general, una *vista* de una estructura de datos compleja (como un *Map* o un *grafo*) es ***otra*** estructura de datos (posiblemente más simple) desde la cual se puede acceder a la original y recorrerla o modificarla, pero de forma tal que si se modifica la estructura original (cambia su tamaño o cambia su contenido) se modifica también la vista ***y viceversa***. Una *vista*, además, es típicamente una estructura de datos *stateless* (*sin estado*): esto significa que esa estructura internamente no almacena datos de ningún tipo (*no es* una copia (compleja o simple) del contenido original en otra estructura) sino que sólo contiene métodos que permiten operar directamente sobre los datos contenidos en la estructura original. Este detalle hace, por ejemplo, que un *iterador* no pueda ser considerado estrictamente una *vista*, ya que si bien un iterador permite acceder y modificar a otra estructura, al mismo tiempo contiene atributos internos que le permiten recordar en qué posición se encuentra el iterador, y eventualmente podría contener una copia de parte o de toda la estructura original. Y finalmente, como es de esperar, una clase que representa una *vista* implementa *Iterator*, por lo que esa vista provee un método para crear y retornar un iterador que en forma directa o indirecta permitirá recorrer la estructura original.

Pues bien: la interface *Map* prevé métodos para la creación de tres tipos de vistas para un map: una primera vista que acceda al conjunto de todas las *claves (keys) y sólo las claves* contenidas en el map (y será manejada con el atributo *keySet* en nuestro caso), una segunda vista que acceda a todos *los pares (key, value)* contenidos en el map (manejada en nuestro caso por el atributo *entrySet*), y una tercera vista que acceda al conjunto de todos los *valores asociados a las claves (values) y sólo a esos valores* (manejada en nuestro caso con el atributo *values*).

Los métodos especificados por *Map* para crear estas vistas son:

- ✓ Set<K> keySet()
- ✓ Set<Map.Entry<K,V>> entrySet()
- ✓ Collection<V> values()

Por lo tanto, una clase que implemente *Map* deberá dar una implementación de cada uno de estos tres métodos (además del resto de los métodos pedidos por *Map*). Para poder implementar estos métodos, hay algo de trabajo que el programador debe hacer. Lo resumimos así:

i. Crear tres *clases internas* que implementen *Set<K>*, *Set<Map.Entry<K, V>>* y *Collection<V>* (ya que cada uno de los métodos debe crear y retornar un objeto de una clase que haya implementado esas interfaces). Esas clases internas serán las que representen a las *vistas* (del mismo modo que para retornar un iterador es común implementar una clase interna que implemente *Iterator* y que represente al iterador). En el caso de nuestra clase *TSBHashtable*, esas clases están implementadas y se llaman respectivamente *KeySet*, *EntrySet* y *ValueCollection* (y hemos seguido aquí el modelo

de implementación de la propia clase *java.util.Hashtable*). La clase *KeySet* es directa y luce así (excluyendo comentarios javadoc y anotaciones):

```
private class KeySet extends AbstractSet<K>
{
   public Iterator<K> iterator()
   {
      return new KeySetIterator();
   }

   public int size()
   {
      return TSBHashtable.this.count;
   }

   public boolean contains(Object o)
   {
      return TSBHashtable.this.containsKey(o);
   }

   public boolean remove(Object o)
   {
      return (TSBHashtable.this.remove(o) != null);
   }

   public void clear()
   {
      TSBHashtable.this.clear();
   }

   private class KeySetIterator implements Iterator<K>
   {
      // índice de la lista actualmente recorrida...
      private int current_bucket;

      // índice de la lista anterior (si se requiere en remove())...
      private int last_bucket;

      // índice del elemento actual en el iterador (el que fue retornado
      // la última vez por next() y será eliminado por remove())...
      private int current_entry;

      // flag para controlar si remove() está bien invocado...
      private boolean next_ok;

      // el valor que debería tener el modCount de la tabla completa...
      private int expected_modCount;

      public KeySetIterator()
      {
         current_bucket = 0;
         last_bucket = 0;
         current_entry = -1;
         next_ok = false;
```

```
    expected_modCount = TSBHashtable.this.modCount;
  }

  @Override
  public boolean hasNext()
  {
    TSBArrayList<Map.Entry<K, V>> t[] = TSBHashtable.this.table;

    if(TSBHashtable.this.isEmpty()) { return false; }
    if(current_bucket >= t.length) { return false; }

    // bucket actual vacío o listo?...
    if(t[current_bucket].isEmpty() || current_entry >= t[current_bucket].size() - 1)
    {
      // ... -> ver el siguiente bucket no vacío...
      int next_bucket = current_bucket + 1;
      while(next_bucket < t.length && t[next_bucket].isEmpty())
      {
        next_bucket++;
      }
      if(next_bucket >= t.length) { return false; }
    }

    return true;
  }

  public K next()
  {
    // control: fail-fast iterator...
    if(TSBHashtable.this.modCount != expected_modCount)
    {
      throw new ConcurrentModificationException("next(): modificación inesperada...");
    }

    if(!hasNext())
    {
      throw new NoSuchElementException("next(): no existe el elemento pedido...");
    }

    TSBArrayList<Map.Entry<K, V>> t[] = TSBHashtable.this.table;

    // se puede seguir en el mismo bucket?...
    TSBArrayList<Map.Entry<K, V>> bucket = t[current_bucket];
    if(!t[current_bucket].isEmpty() && current_entry < bucket.size() - 1) { current_entry++; }
    else
    {
      // si no se puede...
      // ...recordar el índice del bucket que se va a abandonar..
      last_bucket = current_bucket;

      // buscar el siguiente bucket no vacío, que DEBE existir, ya
      // que se hasNext() retornó true...
      current_bucket++;
      while(t[current_bucket].isEmpty())
```

```
            {
               current_bucket++;
            }

            // actualizar la referencia bucket con el núevo índice...
            bucket = t[current_bucket];

            // y posicionarse en el primer elemento de ese bucket...
            current_entry = 0;
         }

         // avisar que next() fue invocado con éxito...
         next_ok = true;

         // y retornar la clave del elemento alcanzado...
         K key = bucket.get(current_entry).getKey();
         return key;
      }

      public void remove()
      {
         if(!next_ok)
         {
            throw new IllegalStateException("remove(): debe invocar a next() antes...");
         }

         // eliminar el objeto que retornó next() la última vez...
         TSBHashtable.this.table[current_bucket].remove(current_entry);

         // quedar apuntando al anterior al que se retornó...
         if(last_bucket != current_bucket)
         {
            current_bucket = last_bucket;
            current_entry = TSBHashtable.this.table[current_bucket].size() - 1;
         }

         // avisar que el remove() válido para next() ya se activó...
         next_ok = false;

         // la tabla tiene un elementon menos...
         TSBHashtable.this.count--;

         // fail_fast iterator: todo en orden...
         TSBHashtable.this.modCount++;
         expected_modCount++;
      }
   }
}
```

Las otras dos clases son similares a esta. Como se ve, la clase *KeySet* no incluye atributos, y cada uno de sus métodos opera directamente con los atributos de la clase contenedora *TSBHashtable* (lo cual obviamente garantiza que si cambia la tabla hash, cambiará también la vista…). La clase *KeySet* provee un método para eliminar un

objeto de la vista, pero ese método simplemente invoca al método *remove()* de la tabla (por lo cual, obviamente, al cambiar la vista está cambiando realmente la tabla…) Finalmente, note que la clase *KeySet* incluye a su vez una *clase interna KeySetIterator* que provee un mecanismo iterador para la vista/tabla: *KeySetIterator* implementa *Iterator*, pero los métodos de esta clase iteran directamente sobre la tabla (y no sobre la vista, que realmente es sólo una fachada).

Note que la clase *KeySetIterator* incluye un atributo *expected_modCount* que se usa para el control *fail fast*: al crearse un objeto iterador, este atributo se inicializa con el valor del atributo *modCount* de la clase *TSBHashtable*, y luego se controla que ambos sigan siendo iguales al invocar a *next()*. Si no lo fuesen (lo cual sólo podría ocurrir en un contexto concurrente) el método *next()* lanza manualmente una excepción de la clase *ConcurrentModificationException*.

ii. Implementar los tres métodos de creación de vistas. La idea (tomada también en forma directa del codigo fuente de la clase *java.util.Hashtable*) es que si una vista es *stateless* y no tendrá atributos ni datos propios, entonces no se justifica crear más de uno de esos objetos por cada clase en una aplicación. La creación de los objetos vista se puede hacer en el momento de invocar por primera vez a cada uno de los tres métodos de *Map*, y dejarlos de ahí en adelante ya creados. Los tres métodos lucen así (en nuestra versión *no thread-safe*):

```
public Set<K> keySet()
{
        if(keySet == null)
        {
                keySet = new KeySet();
                // keySet = Collections.synchronizedSet(new KeySet());
        }
        return keySet;
}

public Set<Map.Entry<K, V>> entrySet()
{
          if(entrySet == null)
          {
                  entrySet = new EntrySet();
                  // entrySet = Collections.synchronizedSet(new EntrySet());
          }
          return entrySet;
}

public Collection<V> values()
{
          if(values==null)
          {
                  values = new ValueCollection();
                  // values = Collections.synchronizedCollection(new ValueCollection());
          }
          return values;
```

```
}
```

Los tres métodos crean respectivamente un objeto de la clase interna que corresponda y lo retornan, pero sólo si el atributo para manejar esos objetos era *null* al momento de invocar al método (en caso contrario, el método retorna el objeto que ya existía).

Las líneas alternativas que figuran comentadas en cada método, muestran la forma en que deberían crearse esos objetos si la clase fuese *thread-safe* (como *java.util.Hashtable*): en ese caso, se recomienda el uso de los métodos *Collections.synchronizedSet()* y *Collections.syncronizedCollection()* quee toman como parámetro un objeto de la clase que cada método debe luego retornar, y crean y retornan otro objeto que envuelve (actúa como objeto *wrapper*) al que se tomó como parámetro.

En rigor de verdad, lo único que hacen los métodos *Collections.synchronizedSet()* y *Collections.synchronizedCollections()* es crear un objeto *wrapper* para el original, de modo que el *wrapper* garantice que el estado del objeto envuelto quede siempre bien definido si se utiliza en un contexto de múltiples hilos de ejecución accediendo al mismo objeto. Como dijimos, nuestra implementación de la clase *TSBHashtable* sigue en ese sentido al modelo de la clase *java.util.HashMap* que no está basada en mantener consistencia en un entorno de ejecución en contextos concurrentes[3] por lo que no nos preocuparemos de estos detalles.

El modelo *TSBHashtable* que acompaña a esta ficha, incluye una clase simple llamada *Test*, con un método *main()* para hacer algunas pruebas muy básicas:

```
public class Test
{
    public static void main(String args[])
    {
        // una tabla "corta" con factor de carga pequeño...
        TSBHashtable<Integer, String> ht1 = new TSBHashtable<>(3, 0.2f);
        System.out.println("Contenido inicial: " + ht1);

        // algunas inserciones...
        ht1.put(1, "Argentina");
        ht1.put(2, "Brasil");
        ht1.put(3, "Chile");
        ht1.put(4, "Mexico");
        ht1.put(5, "Uruguay");
        ht1.put(6, "Perú");
        ht1.put(7, "Colombia");
        ht1.put(8, "Ecuador");
```

[3] El tema es que si se supone que un objeto está *sincronizado* (pensado para ser *thread-safe*) entonces cada bloque de código en el cual se acceda a ese objeto debería estar también definido como *synchronized*, cosa que activa ciertos mecanismos internos de control de la máquina virtual que hacen la ejecución más pesada. Por este motivo existen clases *thread-safe* y clases equivalentes a estas pero que no son *thread-safe*, y el uso de cada una depende del contexto esperado por el programador.

```
            ht1.put(9, "Paraguay");
            ht1.put(10, "Bolivia");
            ht1.put(11, "Venezuela");
            ht1.put(12, "Estados Unidos");
            System.out.println("Luego de algunas inserciones: " + ht1);

            TSBHashtable<Integer, String> ht2 = new TSBHashtable<>(ht1);
            System.out.println("Segunda tabla: " + ht2);

            System.out.println("Tabla 1 recorrida a partir de una vista: ");
            Set<Map.Entry<Integer, String>> se = ht1.entrySet();
            Iterator<Map.Entry<Integer, String>> it = se.iterator();
            while(it.hasNext())
            {
               Map.Entry<Integer, String> entry = it.next();
               System.out.println("Par: " + entry);
            }
        }
    }
```

En el modelo anterior, las líneas remarcadas en rojo muestran en qué forma puede usarse el método *entrySet()* para generar una vista de todos los pares alojados en la tabla *ht1*. El método retorna un objeto que será manejado con la referencia *se*, y ese objeto dispone del método *iterator()* para crear un iterador. El ciclo *while* que aparece luego, recorre la vista/tabla con ese iterador, y muestra todos los pares contenidos en ella.

Los constructores de la clase *TSBHashtable* son los siguientes:

```
public TSBHashtable()
{
   this(5, 0.8f);
}

public TSBHashtable(int initial_capacity)
{
   this(initial_capacity, 0.8f);
}

public TSBHashtable(int initial_capacity, float load_factor)
{
   if(load_factor <= 0) { load_factor = 0.8f; }
   if(initial_capacity <= 0) { initial_capacity = 11; }
   else
   {
      if(initial_capacity > TSBHashtable.MAX_SIZE)
      {
         initial_capacity = TSBHashtable.MAX_SIZE;
      }
   }

   this.table = new TSBArrayList[initial_capacity];
   for(int i=0; i<table.length; i++)
   {
```

```
        table[i] = new TSBArrayList<>();
    }

    this.initial_capacity = initial_capacity;
    this.load_factor = load_factor;
    this.count = 0;
    this.modCount = 0;
}

public TSBHashtable(Map<? extends K,? extends V> t)
{
    this(11, 0.8f);
    this.putAll(t);
}
```

Los dos primeros invocan al tercero de estos constructores, y este úlitmo es el que verdaderamente hace el trabajo: recibe dos parámetros *initial_capacity* y *load_factor*. El primero indica el tamaño con el que debe ser creada la tabla inicialmente, y el segundo informa el factor de carga a considerar para saber cuándo hacer un rehash (que como dijimos, será un número en coma flotante que al multiplicarse por 10, nos dará la longitud promedio límite que admitiremos para las listas de la tabla antes de un rehash.) El constructor inicializa los atributos de la clase, crea el arreglo con *new* y lo inicializa a su vez con listas vacías en cada casilla.

El cuarto constructor se incluye por recomendación surgida de la documentación javadoc: recibe como parámetro un map (supuestamente ya creado), y crea la tabla copiando en ella los elementos del map que entró como parámetro. Para ello, se invoca primero al tercer constructor (creando así una tabla de tamaño inicial 11 y factor de carga 0.8) y luega usa el método *putAll()* (incluido en la misma clase *TSBHashtable*) para hacer la copia. El método *putAll()* es el siguiente:

```
public void putAll(Map<? extends K, ? extends V> m)
{
    for(Map.Entry<? extends K, ? extends V> e : m.entrySet())
    {
        put(e.getKey(), e.getValue());
    }
}
```

Esencialmente, se invoca al método *entrySet()* (que ya vimos) para crear una vista iterable de todos los pares *(key, value)* de la tabla, se itera sobre esa vista y se inserta cada uno de los pares en la tabla actual invocando al método *put()* (que veremos a continuación).

Finalmente, la clase *TSBHastable* implementa el resto de los métodos de la interface *Map*, entre los cuales están *put()* y *get()*. El primero permite agregar un par *(key, value)* en la tabla, y el segundo permite buscar una clave (key) en la tabla, y recuperar el objeto asociado a ella (value) en la tabla:

```
public V get(Object key)
{
        if(key == null) throw new NullPointerException("get(): parámetro null");
```

```
        int ib = this.h(key.hashCode());
        TSBArrayList<Map.Entry<K, V>> bucket = this.table[ib];

        Map.Entry<K, V> x = this.search_for_entry((K)key, bucket);
        return (x != null)? x.getValue() : null;
}

public V put(K key, V value)
{
        if(key == null || value == null) throw new NullPointerException("put(): parámetro null");

        int ib = this.h(key);
        TSBArrayList<Map.Entry<K, V>> bucket = this.table[ib];

        V old = null;
        Map.Entry<K, V> x = this.search_for_entry((K)key, bucket);
        if(x != null)
        {
           old = x.getValue();
           x.setValue(value);
        }
        else
        {
           if(this.averageLength() >= this.load_factor * 10) this.rehash();
           ib = this.h(key);
           bucket = this.table[ib];

           Map.Entry<K, V> entry = new Entry<>(key, value);
           bucket.add(entry);
           this.count++;
           this.modCount++;
        }

        return old;
}

private Map.Entry<K, V> search_for_entry(K key, TSBArrayList<Map.Entry<K, V>> bucket)
{
   Iterator<Map.Entry<K, V>> it = bucket.iterator();
   while(it.hasNext())
   {
      Map.Entry<K, V> entry = it.next();
      if(key.equals(entry.getKey())) return entry;
   }
   return null;
}
```

Ambos métodos toman la clave *key* recibida como parámetro, aplican sobre ella la función *h()* de dispersión (con lo cual obtienen el índice de la lista en la que debe buscar luego al par que contiene a esa clave) y buscan en esa lista (en forma secuencial) un par que contenga a esa clave. El método *get()* retorna el value asociado a esa clave (si existe) o *null* (si no existe la clave). En cuanto al método put(), si la clave buscada por ya existe en la tabla, el método

reemplaza el *value* asociado a ella (y en este caso, el tamaño de la tabla no cambia). Esto implica que sólo si la clave no existe será insertado un par nuevo en la tabla (aumentando su cantidad de objetos). A su vez, *put()* retorna el objeto que anteriormente estaba asociado a la clave (el objeto que fue reemplazado) si es que existía, o bien *null* si la clave no tenía asociado un *value*. Tanto *get()* com *put()* utilizan los servicios de un método auxiliar privado llamado *search_for_entry()*, el cual toma como parámetro una clave y una lista en la cual debe buscar la clave. Si encuentra esa clave en la lista, retorna el objeto *Entry* (el par *(key, value)*) que contiene a esa clave, o retorna *null* si tal clave no existe en ningún par de la lista.

En cuanto a la función *h()* de dispersión, la clase *TSBHahtable* define (por comodidad) cuatro versiones privadas, que se muestran a continuación:

```
private int h(int k)
{
   return h(k, this.table.length);
}

private int h(K key)
{
   return h(key.hashCode(), this.table.length);
}

private int h(K key, int t)
{
   return h(key.hashCode(), t);
}

private int h(int k, int t)
{
   if(k < 0) k *= -1;
   return k % t;
}
```

Como se puede ver, la que realmente hace el trabajo es la última versión, que toma como parámetro un valor o clave *k* de tipo entero y retorna el resto de dividir a esa clave por el valor *t* que también toma como parámetro (y que a todos los efectos prácticos debe entenderse como el tamaño de la tabla en la cual se quiere agregar la clave). Note que el método comprueba si k es negativo (lo cual podría ocurrir…) y en ese caso simplemente cambia su signo multiplicándolo por -1.

El último de los métodos realmente importantes de la clase (y que no está especificado por *Map* sino que es propio y específico de *TSBHashtable*) es el método *rehash()* que se muestra a continuación:

```
protected void rehash()
{
   int old_length = this.table.length;

   // nuevo tamaño: doble del anterior, más uno para llevarlo a impar...
   int new_length = old_length * 2 + 1;
```

```
        // no permitir que la tabla tenga un tamaño mayor al límite máximo...
        // ... para evitar overflow y/o desborde de índices...
        if(new_length > TSBHashtable.MAX_SIZE)
        {
           new_length = TSBHashtable.MAX_SIZE;
        }

        // crear el nuevo arreglo con new_length listas vacías...
        TSBArrayList<Map.Entry<K, V>> temp[] = new TSBArrayList[new_length];
        for(int j = 0; j < temp.length; j++) { temp[j] = new TSBArrayList<>(); }

        // notificación fail-fast iterator... la tabla cambió su estructura...
        this.modCount++;

        // recorrer el viejo arreglo y redistribuir los objetos que tenia...
        for(int i = 0; i < this.table.length; i++)
        {
                // entrar en la lista numero i, y recorrerla con su iterador...
                Iterator<Map.Entry<K, V>> it = this.table[i].iterator();
                while(it.hasNext())
                {
                        // obtener un objeto de la vieja lista...
                        Map.Entry<K, V> x = it.next();

                        // obtener su nuevo valor de dispersión para el nuevo arreglo...
                        K key = x.getKey();
                        int y = this.h(key, temp.length);

                        // insertarlo en el nuevo arreglo, en la lista numero "y"...
                        temp[y].add(x);
                }
        }

        // cambiar la referencia table para que apunte a temp...
        this.table = temp;
    }
```

Este método expande el tamaño del arreglo de soporte y lo lleva al "doble más uno" de su tamaño anterior (esto es para asegurar que ese nuevo tamaño sea impar y por lo tanto tenga mejores probabilidades de evitar colisiones). Una vez que el nuevo arreglo ha sido creado, se recorre el arreglo anterior lista por lista, mediante un iterador para cada lista, y cada uno de los pares que contenía el viejo arreglo se *vuelve a dispersar* en el nuevo (esto es, se calcula con *h()* el nuevo índice que tendrá ese par en la nueva tabla). Como vimos, no se copian directamente las viejas listas en la misma casilla de la nueva tabla, ya que al haber cambiado el tamaño del arreglo cambió también el índice de entrada de cada par.

Dejamos para el estudiante el análisis detallado del resto de los métoso de la clase incluidos en el código fuente del modelo.