Relación de Ejercicios 5 (Tablas Hash)

Para realizar estos ejercicios necesitarás crear diferentes ficheros tanto en Haskell como en Java. En cada caso crea un nuevo fichero (con extensión hs para Haskell y java para Java). Añade al principio de tu fichero la siguiente cabecera, reemplazando los datos necesarios:

```
--------------------------------------------------------------------------------
-- Estructuras de Datos. 2° Curso. ETSI Informática. UMA
--
-- (completa y sustituye los siguientes datos)
-- Titulación: Grado en Ingeniería ..... [Informática | del Software | de Computadores].
-- Alumno: APELLIDOS, NOMBRE
-- Fecha de entrega:  DIA | MES | AÑO
--
-- Relación de Ejercicios 5. Ejercicios resueltos: ..........
--
--------------------------------------------------------------------------------
import Test.QuickCheck   (Cuando se trate de Haskell)
```

1. (Java) Lee y estudia detenidamente el código completo de la clase `SeparateChainingHashTable` que implementa el interfaz `HashTable` que aparece en las transparencias (el código completo está disponible en el Campus Virtual). Presta especial atención a la implementación de los métodos `delete`, `rehashing` y al iterador que recorre la tabla.

2. (Java) Añade otro iterador a la clase `SeparateChainingHashTable` pero que recorra la tabla devolviendo sus valores. Define el método `values` que devuelva una instancia de este interador.

3. (Java) Implementa una clase genérica `Tuple2<A,B>` para representar pares de dos valores (componentes) de tipos A y B. Además del constructor, define dos métodos para devolver la primera y segunda componentes de un par:

```
public A _1();
public B _2();
```

Redefine también el método `toString` para visualizar los elementos del par como una cadena de caracteres de la forma "`Tuple2(x,y)`"; define también los métodos `equals` (que devuelve `true` si las componentes de los objetos a comparar son iguales dos a dos) y `hashCode` (devuelve el código hash de un par combinando de forma adecuada los códigos de las componentes).

4. (Java) Utiliza la clase anterior para añadir otro iterador `keysValues` a la clase `SeparateChainingHashTable` de forma que el método `next()` devuelva información de los elementos de la tabla en forma de par:

```
private class KeysValuesIter extends NodesIter
              implements Iterator<Tuple2<K,V>> { ... }

public Iterable<Tuple2<K,V>> keysValues () { ... }
```

5. (Java) Desarrolla una implementación alternativa de la clase `SeparateChainingHashTable<K,V>` pero que, internamente, en lugar de nodos utilice una tabla de listas encadenadas:

```
public class SeparateChainingHashTable<K,V> implements HashTable<K,V> {
       private List<<Pair<K,V>> table[];
       private int size; // number of elements inserted in table
       private double maxLoadFactor;
       . . .
}
```

6. (Java) El *Java Collections Framework* proporciona en la clase `java.util.Hashtable` una implementación de tablas hash. Lee la documentación y estudia el código de esta clase.

7. (Haskell o Java) **(Ejercicio 3.4.4 de Sedgewick & Wayne, 2011, p.480)** Escribe un programa para encontrar los menores valores de *a* y *M* (con *M* tan pequeño como sea posible) de modo que la función hash (*a* * *k*) % *M*, que transforma la *k*-ésima letra del alfabeto en un índice de una tabla de tamaño *M*, produzca distintos valores (ninguna colisión) para las claves `S E A R C H X M P L`. Una función con tal propiedad se denomina una función de hash perfecta.

8. (Haskell o Java) **(Ejercicio 3.4.36 de Sedgewick & Wayne, 2011, p.485**) (*List length range*) Escribe un programa que inserte N claves enteras aleatorias en una tabla de tamaño *N*/100 usando *separate chaining* y localice la longitud más corta de las listas de la tabla. Analiza estos valores para $N = 10^3, 10^4, 10^5$ y 10^6.

9. Implementa la interfaz `Bag` (definida en los ejercicios del tema 3) usando una tabla hash, de forma que las claves correspondan a los elementos de la bolsa (o multiconjunto) y los valores al número de ocurrencias de cada elemento.

10. Implementa la interfaz `Dictionary` (definida en las transparencias del tema 4) usando una tabla hash.

11. (Java) Una forma eficiente de representar un conjunto de números naturales es a través de los llamados *Bitsets*. La idea es usar una tabla de n bytes para representar un subconjunto de valores del rango { 0 .. (8xn) – 1 } (recordemos que cada byte está formado por 8 bits) de forma que el *i*-ésimo bit del elemento b la tabla es 1 si y solo si el número *8b+i* está en el conjunto.

a) Implementa una clase `Bitset` que incluya los siguientes métodos:

```
public class Bitset {
  public Bitset(int n); // crea un bitset de n bytes
  public void insert(int x);  // precondición: 0 ≤ x < 8*n. Inserta x
  public void delete(int x);  // precondición: 0 ≤ x < 8*n. Elimina x
  public boolean isElem(int x);  //
  public boolean isEmpty(); //
  public String toString()
}
```

b) Añade a la cabecera de la clase `Bitset extends Iterable<Integer>` y define un iterador que devuelva los elementos del conjunto en orden ascendente.

c) Modifica la implementación para que se produzca un redimensionado de la tabla si el valor a añadir es ≥ 8*n.

d) Añade métodos para calcular la unión, intersección y diferencia de dos conjuntos.

12. (Java) Implementa una tabla hash usando la técnica *Linear Probing* (tal como se describe en las últimas transparencias del tema 5). La clase debe implementar la siguiente interfaz:

```
public interface HashTable<K, V> extends Iterable<K> {
  public boolean isEmpty();
  public int size();
  public void insert(K key, V value);
  public V search(K key);
  public boolean isElem(K key);
  public void delete(K key);
  Iterable<K> keys();
  Iterable<V> values();
  Iterable<Tuple2<K,V>> keysValues();
}
```

Usa las siguientes variables de instancia y constructor de la clase:

```
public class LinearProbingHashTable<K,V> implements HashTable<K,V> {
  private K keys[];
  private V values[];
  private int size;
  private double maxLoadFactor;

  public LinearProbingHashTable(int numCells, double loadFactor) {
        keys = (K[]) new Object[numCells];
        values = (V[]) new Object[numCells];
        size = 0;
        maxLoadFactor = loadFactor;
  }
}
```

Como primer paso, comienza definiendo el método:

```
private int searchIdx(K key)
```

que toma una clave y devuelve la posición de la tabla donde debemos insertar un elemento con dicha clave, de acuerdo al método de prueba lineal. Para memorizar pares de claves y valores, usaremos dos tablas (`keys` y `values`); si la posición devuelta por el método `searchIdx` correspondiente a una clave k es p, k deberá memorizarse en `keys[p]` y el correspondiente valor en `values[p]`. Si, tras un número de inserciones, el factor de carga sobrepasa el límite `maxLoadFactor`, las tablas deben redimensionarse a través del método:

```
private void rehashing() {
  // computamos un nuevo tamaño de las tablas
  int newCapacity = HashPrimes.primeDoubleThan(keys.length);
  K oldKeys[] = keys;
  V oldValues[] = values;
```

```
    keys = (K[]) new Object[newCapacity];
    values = (V[]) new Object[newCapacity];

    // reinsertamos los elementos en las nuevas tablas
    for(int i=0; i<oldKeys.length; i++)
          if(oldKeys[i] != null) {
                int newIdx = searchIdx(oldKeys[i]);
                keys[newIdx] = oldKeys[i];
                values[newIdx] = oldValues[i];
          }
}
```

Para implementar la operación `delete`, debemos localizar primero la posición correspondiente `p` en la tabla de claves, y asignar `null` a las posiciones `keys[p]` y `values[p]`. A continuación, debemos *trasladar* (borrar y reinsertar) los elementos posteriores para no dejar *huecos*.

13. (Ejercicio 3.4.13 de Sedgewick & Wayne, p.481) ¿Cuál de las siguientes situaciones conduce a una ejecución en tiempo lineal para una búsqueda de un elemento arbitrario en una tabla hash usando linear-probing?

a) Todas las claves tienen el mismo valor hash.
b) Todas las claves tienen distinto valor hash.
c) Todas las claves tienen un valor hash par.
d) Todas las claves tienen valores hash pares y diferentes.

14. (Ejercicio 3.4.18 de Sedgewick & Wayne, p.482) Añade un constructor para la clase `SeparateChainingHashTable` que permita especificar el número promedio de colisiones tolerados para una búsqueda. Para controlar esto, debes redimensionar el array (usando siempre un tamaño primo) de forma que el tamaño promedio de las listas sea menor que el valor especificado.

15. (Ejercicio 3.4.20 de Sedgewick & Wayne, p.482) Agrega un método a la clase `LinearProbingHashTable` que calcule el coste promedio de una búsqueda, suponiendo que la probabilidad de buscar las disitintas claves es uniforme (misma probabilidad).

16. El Problema del Parking de Knuth (3.4.43, Sedgewick & Wayne p. 485) Escribe programas para comprobar experimentalmente la siguiente hipótesis: el número de comparaciones necesarias para insertar M claves aleatorias sobre una tabla de tamaño M usnado el método *linear probing* es $\sim M^{3/2}\sqrt{\pi/2}$.