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. 2º 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 (dicho código está disponible en el Campus Virtual). Presta atención especial a la implementación de los métodos `delete`, `rehashing` y el iterador que recorre la tabla.

2. (Java) Añade otro iterador (`values`) a la clase `SeparateChainingHashTable` pero que recorra la tabla devolviendo los valores. Define el método `values` que devuelva una instancia del 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 string de la forma "Tuple2(x,y)"; define también los métodos `equals` (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 () { ... }
```

4B. (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;
      . . .
```

```
}
```

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

6. (Haskell o Java) **(Ejercicio 3.4.4 de Sedgewick & Wayne, 2011, p.480)** Escribe una función para encontrar los menores valores de `a` y `m` tales que la función hash

```
hash :: Char -> Int
hash k = a * (ord k − ord 'A') `mod` m
```

no produzca colisiones para valores diferentes de la lista de claves

```
keys = "SEARCHEXAMPLE"
```

6B. Usando la función hash anterior, inserta las diferentes claves de la palabra `"EAASSYQUESTION"` en este mismo orden utilizando prueba lineal sobre una tabla de tamaño m = 16. Idem al anterior pero para m=10.

7. (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.

8. Implementa la interfaz `Bag` (práctica 3 del tema 3) usando una tabla hash de forma que las claves correspondan a los elementos del saco (o multiconjunto) y los valores al número de ocurrencias de cada elemento.

9. Implementa la interfaz `Dictionary` (transparencia 170 del tema 4) usando una tabla hash.

10. (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 extend Iterable<Integer>` y define un iterador que devuelva los elementos del conjunto en el orden natural.

c) Modifica la implementación para que se produzca una redimensión de la tabla si el elemento a añadir es ≥ 8*n.

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

11. (Java) Implementa los métodos correspondientes si usamos *Linear Probing* como se describe en las últimas transparencias del tema 5. La clase debe implementar la interfaz siguiente:

```
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;
  }
}
```

Antes de todo, define el método:

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

que toma una clave y devuelve la posición donde debemos insertar un elemento con tal clave utilizando prueba lineal. Para memorizar pares de claves y valores usaremos dos tablas; 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 reasignarse 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`, primeramente debemos localizar la posición correspondiente p en la tabla de claves, y asignar `null` a las posiciones `keys[p]` y `values[p]`, y a continuación *trasladar* (borrar y reinsertar) los elementos posteriores para no dejar *huecos*.

12. (Ejercicio 3.4.13 de Sedgewick & Wayne, p.481) ¿Cuál de las siguientes situaciones conduce a una ejecución en tiempo lineal para una búsqueda aleatoria 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.

13. (Ejercicio 3.4.18 de Sedgewick & Wayne, p.482) Añada un constructor para la clase SeparateChainingHashTable que permita especificar un número máximo promedio de colisiones. Debe redimensionar la tabla si se sobrepasa el máximo de colisiones, tomando un tamaño primo.

14. (Ejercicio 3.4.20 de Sedgewick & Wayne, p.482) Agregar un método a LinearProbingHashTable que calcule el coste promedio de una búsqueda, suponiendo que cada clave será buscada con la misma probabilidad.

15. (Ejercicio 3.4.26 de Sedgewick & Wayne, p.483) (*lazy delete in linear probing*) Añade a la clase LinearProbingHashTable un método `delete()` que elimine un par clave-valor estableciendo el valor a `null` (sin dejar el hueco); éste será eliminado más tarde en una operación de redimensionado. Debes definir algún parámetro que permita decidir cuándo realizar el redimensionado. Nota: debes permitir escribir encima de un `null` durante la operación `insert(K key, V value)`. Asegúrate de que el programa contabilice el número de elementos a null y el número de posiciones vacías para decidir el redimensionado.

16. Escribe programas para realizar pruebas aleatorias que computen tiempos y colisiones de los diferentes métodos.

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