Resolución del Segundo Parcial de EDA (27 de Abril de 2015) Puntuación 2,4

1.- Se dispone de un `Map` `m` en el que cada clave es el nombre de un tifón y su valor asociado es el año en el que ocurrió. Para poder conocer cuántos tifones de los registrados en este Map se han producido en cada uno de los años que aparecen en él, **se pide** diseñar un método con perfil

```
public static Map<Integer, Integer> tifones(Map<String, Integer> m)
```

tal que, dado `m`, devuelva un `Map` en el que cada clave sea un año y su valor asociado sea el número de tifones que ocurrieron en él. Además, como ambos Maps se implementan con Tablas Hash en las que el coste promedio de insertar y recuperar es Θ(1), el coste de `tifones` debe ser estrictamente del orden de la talla de `m` (Θ(`m.talla()`)). **(0.6 puntos)**

```
public static Map<Integer, Integer> tifones(Map<String, Integer> m) {
   Map<Integer, Integer> res = new TablaHash<Integer, Integer>(m.talla());
   ListaConPI<String> l = m.claves();
   l.inicio();
   while (!l.esFin()) {
      String tifon = l.recuperar();
      Integer anyo = m.recuperar(tifon);
      Integer frec = res.recuperar(anyo);
      if (frec == null) res.insertar(anyo, 1); else res.insertar(anyo, ++frec);
   }
    return res;
}
```

2.- En la clase `TablaHash`, **se pide** diseñar un método `colisionanCon` que, con el menor coste posible, devuelva una Lista Con Punto de Interés con todas las claves de una Tabla Hash que colisionan con una dada `c`. **(0.6 puntos)**

- Si las cubetas se implementan con Listas Con PI:

```
public ListaConPI<C> colisisonanCon(C c) {
   ListaConPI<C> l = new LEGListaConPI<C>();
   if (recuperar(c) == null) return l;
   int pos = indiceHash(c);
   ListaConPI<EntradaHash<C, V>> aux = elArray[pos];
   for (aux.inicio(); !aux.esFin(); aux.siguiente()) {
     C colisiona = aux.recuperar().clave;
     if (!colisiona.equals(c) l.insertar(colisiona);
   }
   return l;
}
```

- Si las cubetas se implementan con Listas Enlazadas Directas:

```
public ListaConPI<C> colisisonanCon(C c) {
   ListaConPI<C> l = new LEGListaConPI<C>();
   if (recuperar(c) == null) return l;
   int pos = indiceHash(c);
   EntradaHash<C, V>> aux = elArray[pos];
   for (; aux != null; aux = aux.siguiente) {
     C colisiona = aux.clave;
     if (!colisiona.equals(c) l.insertar(colisiona);
   }
   return l;
}
```

3.- Se pide diseñar un método genérico y estático `sucesor` que devuelva el sucesor de un dato `k` en una Cola de Prioridad `cP`, implementada con un Montículo Binario Minimal; si el sucesor de `k` no está en `cP` el método devuelve `null` para advertirlo. Además, dicho método tiene que usar una Pila como estructura de datos auxiliar, pues al terminar su ejecución `cP` debe contener los mismos elementos que tenía antes de invocarlo y se quiere que su coste sea el mejor posible (Ω (1) y O(x·`log`x), siendo x la talla de `cP`); queda a criterio del programador que dicha Pila sea la Pila de la Recursión o una implementada mediante un `ArrayPila`. **(0.6 puntos)**

- Si se utiliza la Pila de la Recursión:

```
public static <E extends Comparable <E>> E sucesor(ColaPrioridad<E> cP, E k) {
   if (cP.esVacia()) return null;
   E min = cP.recuperarMin();
   if (min.compareTo(k) > 0) res = min;
   else {
        min = cP.eliminarMin();
        res = succesor(cP, k);
        cP.insertar(min);
   }
   return res;
}
```

- Si se utiliza una Pila auxiliar:

```
public static <E extends Comparable <E>> E sucesor(ColaPrioridad<E> cP, E k) {
   E res = null;
   Pila<E> aux = new ArrayPila<E>();
   while (!cP.esVacia() && cP.recuperarMin().compareTo(k) <= 0) {
        E min = cP.eliminarMin();
        aux.apilar(min);
   }
   if (!cP.esVacia()) res = cP.recuperarMin();
   while (!aux.esVacia()) cP.insertar(aux.desapilar());
   return res;
}
```

4.- Un alumno ha ideado el siguiente "truco" para conseguir acceder al máximo de un Heap Minimal en tiempo constante: mantenerlo siempre en la última posición del Heap, i.e. en `talla`. Pero, obviamente, ello implica un cambio a la hora de insertar un nuevo elemento `e` en el Heap.

Se pide implementar en la clase `MonticuloBinario` este nuevo método de inserción, con coste O(`log talla`) y Ω (1). **(0.6 puntos)**

```
public void insertarNuevo(E e) {
    if (talla == 0 || e.compareTo(elArray[talla]) >= 0)
       insertar(e);
    else {
       E max = elArray[talla--];
       insertar(e);
       insertar(max);
    }
}
```

ANEXO

Las interfaces `ListaConPI`**,** `Map` **y** `ColaPrioridad` **del paquete** `modelos`**.**

```
public interface ListaConPI<E> {
  void insertar(E e);
  /** SII !esFin() */ void eliminar();
  void inicio();
  /** SII !esFin() */ void siguiente();
  void fin();
  /** SII !esFin() */ E recuperar();
  boolean esFin();
  boolean esVacia();
  int talla();
}
```

```
public interface Map<C, V> {
       V insertar(C c, V v);
       V eliminar(C c);
       V recuperar(C c);
       boolean esVacio();
       int talla();
       ListaConPI<C> claves();
}
```

```
public interface ColaPrioridad<E extends Comparable<E>> {
   void insertar(E e);
  /** SII !esVacia() */ E  eliminarMin();
  /** SII !esVacia() */ E  recuperarMin();
  boolean esVacia();
}
```

Las clases `TablaHash` **y** `EntradaHash` del paquete `deDispersion`**.**

- Implementación de **cubetas** con **Listas Con Punto de Interés:**

```
class EntradaHash<C, V> {
    C clave; V valor;
    public EntradaHash(C clave, V valor) { this.clave = clave; this.valor = valor; }
}
public class TablaHash<C, V> implements Map<C, V> {
    protected ListaConPI<EntradaHash<C,V>>[] elArray; protected int talla;
    protected int indiceHash(C c) {…}
    public TablaHash(int tallaMaximaEstimada) {…}
    public V recuperar(C c) {…}
    public V eliminar(C c) {…}
    public V insertar(C c, V v) {…}
    public final double factorCarga() {…}
    public final boolean esVacio() {…}
    public final int talla() {…}
    public final ListaConPI<C> claves() {…}
    … //otros métodos
}
```

- Implementación de **cubetas** con **Listas Directas:**

```
class EntradaHash<C, V> {
    C clave; V valor;
    EntradaHash<C, V> siguiente;
    public EntradaHash(C clave, V valor, EntradaHash<C, V> siguiente){
          this.clave = clave; this.valor = valor; this.siguiente = siguiente;
    }
}
public class TablaHash<C, V> implements Map<C, V> {
    protected EntradaHash<C,V>>[] elArray; protected int talla;
    … // Los nombres de los métodos coinciden con los de la anterior clase TablaHash
}
```

La clase `MonticuloBinario` del paquete `jerarquicos`**.**

```
public class MonticuloBinario<E extends Comparable<E>> implements ColaPrioridad<E> {
    protected E elArray[]; protected static final int CAPACIDAD_POR_DEFECTO = 11;
    protected int talla;
    @SuppressWarnings("unchecked") public MonticuloBinario() {…}
    public void insertar(E e) {…}
    /** SII !esVacia() */ public E eliminarMin {…}
    /** SII !esVacia() */ public E recuperarMin(){…}
    … // Otros métodos
}
```