Resolución de la Recuperación del Segundo Parcial de EDA (19 de Junio de 2015 - 2,4 puntos)

1.- Dados dos arrays de palabras (`String`) no vacíos, `v1` y `v2`, **se pide** diseñar un método estático que devuelva un (nuevo) array que contenga aquellas palabras de `v1` que no estén en `v2`, o un array vacío (de longitud 0) si no existe ninguna palabra que cumpla dicha condición. Para que la implementación de este método sea la más eficiente posible, se deberá usar como estructura auxiliar un `Map` implementado mediante una `TablaHash`. **(0.6 puntos)**

```
public static String[] diferencia(String[] v1, String[] v2) {
   Map<String, String> m = new TablaHash<String, String>(v1.length);
   for (String s: v1) m.insertar(s, s);
   for (String s: v2) m.eliminar(s);
   String[] res = new String[m.talla()];
   ListaConPI<String> claves = m.claves(); int i = 0;
   for (claves.inicio(); !claves.esFin(); claves.siguiente())
       res[i++] = claves.recuperar();
   return res;
}
```

Alternativamente, sin usar el método `claves()` de `Map` ...

```
public static String[] diferencia(String[] v1, String[] v2) {
   Map<String, String> m = new TablaHash<String, String>(v2.length);
   for (String s: v2) m.insertar(s, s);
   ListaConPI<String> aux = new LEGListaConPI<String>();
   for (String s: v1) if (m.recuperar(s) == null) aux.insertar(s);
   String[] res = new String[aux.talla()]; int i = 0;
   for (aux.inicio(); !aux.esFin(); aux.siguiente())
       res[i++] = aux.recuperar();
   return res;
}
```

2.- Se ha definido la clase `TablaHashNotas` como una `TablaHash` cuyas claves son los nombres de los alumnos de una asignatura y cuyos valores son sus correspondientes notas:

```
public class TablaHashNotas extends TablaHash<String, Double> { … }
```

Se pide diseñar un método en esta clase que devuelva una `ListaConPI` con los nombres de los alumnos cuya nota esté dentro de un intervalo real `[min, max]` dado. Para que la implementación de este método sea la más eficiente posible, solo se podrá hacer uso de los atributos de la clase `TablaHash`. **(0.6 puntos)**

- Si las cubetas se implementan con Listas Con PI:

```
public ListaConPI<String> alumnosConNota(double min, double max) {
   ListaConPI<String> res = new LEGListaConPI<String>();
   for (int i = 0; i < elArray.length; i++) {
      ListaConPI<EntradaHash<String, Double>> aux = elArray[i];
       for (aux.inicio(); !aux.esFin(); aux.siguiente()) {
           EntradaHash<C, V> e = aux.recuperar();
           if (e.valor >= min && e.valor <= max) res.insertar(e.clave);
       }
   }
   return res;
}
```

- Si las cubetas se implementan con Listas Enlazadas Directas:

```
public ListaConPI<String> alumnosConNota(double min, double max) {
   ListaConPI<String> res = new LEGListaConPI<String>();
   for (int i = 0; i < elArray.length; i++) {
       EntradaHash<String, Double> aux = elArray[i];
       for (; aux != null; aux = aux.siguiente)
           if (aux.valor >= min && aux.valor <= max) res.insertar(aux.clave);
   }
   return res;
}
```

3.- Dados una `ColaPrioridad cP` y dos elementos `x` e `y`, **se pide** diseñar un método genérico y estático que, de la manera más eficiente posible, cambie todos aquellos elementos de `cP` que coincidan con `x` por el elemento `y`. Para ello, dicho método deberá usar una Pila como estructura de datos auxiliar, quedando a criterio del programador que dicha Pila sea la Pila de la Recursión o una implementada mediante un `ArrayPila`. **(0.6 puntos)**

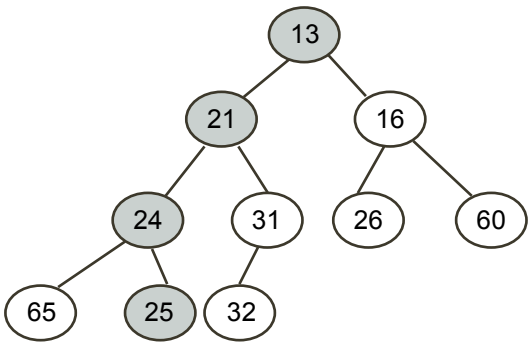
- Si se utiliza la Pila de la Recursión:

```
public static <E extends Comparable<E>> void cambiar(ColaPrioridad<E> cP, E x, E y) {
   if (!cp.esVacia()) {
      E min = cp.eliminarMin();
      int cmp = min.compareTo(x);
      if (cmp <= 0) cambiar(cp, x, y);
      if (cmp == 0) cp.insertar(y); else cp.insertar(min);
   }
}
```

- Si se utiliza una Pila auxiliar:

```
public static <E extends Comparable<E>> void cambiar(ColaPrioridad<E> cP, E x, E y) {
   Pila<E> aux = new ArrayPila<E>();
   while (!cp.esVacia() && cP.recuperarMin().compareTo(x) <= 0) {
         E min = cp.eliminarMin();
         int cmp = min.compareTo(x);
         if (cmp == 0) aux.apilar(y); else aux.apilar(min);
   }
   while (!aux.esVacia()) cP.insertar(aux.desapilar());
}
```

4.- Se pide implementar en la clase `MonticuloBinario` un nuevo método que, con el menor coste posible, devuelva una `ListaConPI` con los elementos del camino que va de la Raíz de un Montículo no vacío a la menor de sus Hojas. Por ejemplo, para el Montículo de la siguiente figura, la Lista a devolver sería `[13, 21, 24, 25]`. **(0.6 puntos)**




```
public ListaConPI<E> caminoALaMenorHoja() {
   ListaConPI<E> res = new LEGListaConPI<E>();
   int posMenor = talla/2 + 1;
   for (int i = talla/2 + 2; i <= talla; i++)
       if (elArray[i].compareTo(elArray[posMenor]) < 0) posMenor = i;
   while (posMenor >= 1) {
         res.inicio();
         res.insertar(elArray[posMenor]);
         posMenor = posMenor / 2;
   }
   return res;
}
```

ANEXO

Las interfaces `ListaConPI`, `Map`, `Pila` **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 Pila<E> {
   void apilar(E e);
   /** SII !esVacia() */ E desapilar();
   /** SII !esVacia() */ E tope();
   boolean esVacia();
}
```

```
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
}
```