EXÁMENES SEGUNDO PARCIAL

1.

Se tiene la siguiente clase `Alumno`:

```
class Alumno {
  private Integer dni;
  private String apellidos;

  public Alumno(Integer dni, String apellidos) {
    this.dni = dni;
    this.apellidos = apellidos;
  }

  public Integer getDni() { return dni; }
  public String getApellidos() { return apellidos };
}
```

(a) Se pide: Implementar el método

```
public static Alumno [] ordenarAlumnos (Alumno [] alumnos)
```

que toma como parámetro un array de objetos de tipo `Alumno` que no será `null` y cuyos elementos tampoco serán `null` y devuelve un *nuevo* array del mismo tamaño que el array `alumnos` con los alumnos ordenados en orden creciente en función de su DNI. El método debe devolver el resultado en un nuevo array y **no debe modificar** el parámetro `alumnos`. Se recomienda utilizar una cola con prioridad usando el interfaz `PriorityQueue<K,V>` (disponible en el apéndice de este examen) para ordenar los elementos. Puede asumirse que se dispone de la clase `SortedListPriorityQueue<K,V>` que implementa el interfaz `PriorityQueue<K,V>`.

(b) Se pide: Implementar el método `compareTo` en la clase `Alumno` para que implemente el interfaz `Comparable<Alumno>` estableciendo un orden creciente de acuerdo al número del DNI. La definición de la clase alumno ahora sería:

```
class Alumno implements Comparable<Alumno> {...}
```

```
public static Alumno[] ordenarAlumnos(Alumno[] alumnos) {
  Alumno[] res = new Alumno[alumnos.length];
  PriorityQueue<String,Alumno> cola =
       new SortedListPriorityQueue<String,Alumno>();
  for(Alumno a: alumnos)
    cola.enqueue(a.getDni(), a);
  int pos = 0;
  while(!cola.isEmpty())
    res[pos++] = cola.dequeue().getValue();
  return res;
}

public static int compareTo(Alumno a) {
  return this.dni - a.getDni();
}
```

2. `Set<E>` es un interfaz que define las operaciones de un conjunto finito de elementos de clase `E`. Un conjunto es una colección de elementos en la que no existe orden y en la que no hay elementos repetidos. Se pide: implementar la clase `MiSet<E>` que implemente el interfaz `Set<E>`, mostrado a continuación, utilizando internamente un atributo de tipo `Map<E,E>` (no está permitido añadir otros atributos):

```
public interface Set<E> {
  /** Devuelve true si el conjunto está vacío y false en caso contrario. */
  public boolean isEmpty();
  /** Devuelve el número de elementos del conjunto. */
  public int size();
  /** Añade el elemento ''elem'' al conjunto. Devuelve true si el
   * elemento ya estaba contenido en el conjunto y false en caso
   * contrario. En caso de que ''elem'' sea null el método debe lanzar
   * la excepción IllegalArgumentException y no se a˜nade al conjunto. */
  public boolean add(E elem);
  /** Elimina ''elem'' del conjunto. Devuelve true si el elemento
   * estaba en el conjunto y ha sido eliminado y false en caso
   * contrario. En caso de que ''elem'' sea null el método debe
   * lanzar la excepción IllegalArgumentException. */
  public boolean remove(E elem);
  /** Devuelve true si el elemento ''o'' está en conjunto. En caso
   * de que ''elem'' sea null el método debe devolver false. */
  public boolean contains(Object o);
  /** Devuelve todos los elementos contenidos en el conjunto en una
   * estructura Iterable. El orden de los elementos en el Iterable no
   * es relevante. */
  public Iterable<E> getElements();
  /** Devuelve un nuevo conjunto intersección, que contiene los elementos
   * que están simultáneamente en los conjuntos ''set'' y ''this''.
   * El conjunto ''set'' nunca será null */
  public Set<E> intersection(Set<E> set);
}
```

Se dispone de la clase `HashTableMap<K,V>`, que implementa el interfaz `Map<K,V>` y que dispone de un constructor sin parámetros para crear un `Map` vacío.

¡¡IMPORTANTE!! En la descripción de los interfaces entregada en el examen hay una errata en el interfaz `Map<E,E>`: Falta el método:

```
/** Returns true if the map contains an entry with the key argument,
 * and false otherwise. */
public boolean containsKey(Object key) throws InvalidKeyException;
```

```
public class MiSet<E> implements Set<E> {
  private Map<E,E> elems;

  public MiSet() {
    elems = new HashTableMap<E,E>();
  }

  public boolean isEmpty(){
    return elems.isEmpty();
  }

  public int size() {
    return elems.size();
  }

  public boolean add(E elem) {
    if(elem == null)
      throw new IllegalArgumentException();
    return elems.put(elem,elem) != null;
  }

  public boolean remove(E elem) {
    if(elem == null)
      throw new IllegalArgumentException();
    return elems.remove(elem)!=null;
  }

  public boolean contains(Object o) {
    return o!=null && elems.containsKey(o);
  }

  public Iterable<E> getElements() {
    return elems.keys();
  }

  public Set<E> intersection(Set<E> set) {
    Set<E> interseccion = new MiSet<E>();

    Iterator<E> it = this.getElements().iterator();
    E e;
    while(it.hasNext()) {
      e = it.next();
      if(set.contains(e))
        interseccion.add(e);
    }

    return interseccion;
  }
}
```

```
for(E e: this.getElements()) {
  if(set.contains(e))
    interseccion.add(e);
}
```

OPCIÓN 1: usando iteradores
OPCIÓN 2: usando for-each

3.

Se pide: Implementar en Java el método

```
public static <E> PositionList<E> ancestros(Tree<E> t, Position<E> p)
```

que toma como parámetro un árbol no vacío `t` y un nodo `p` de dicho árbol. El método debe devolver como resultado una lista de posiciones con los elementos en los ancestros propios de `p`. El interfaz `Tree<E>` está disponible en el Apéndice A.1.

```
public static<E> PositionList<E> ancestros(Tree<E> t, Position<E>
p){
  PositionList<E> ancestros = new NodePositionList<E>();

  if(p!=null) {
    while(!p.isRoot()) {
      p = t.parent(p);
      ancestros.addLast(p.element());
    }
  }

  return ancentros;
}
```

4. Indicar la complejidad en caso peor del método `ancestros` de la pregunta anterior.

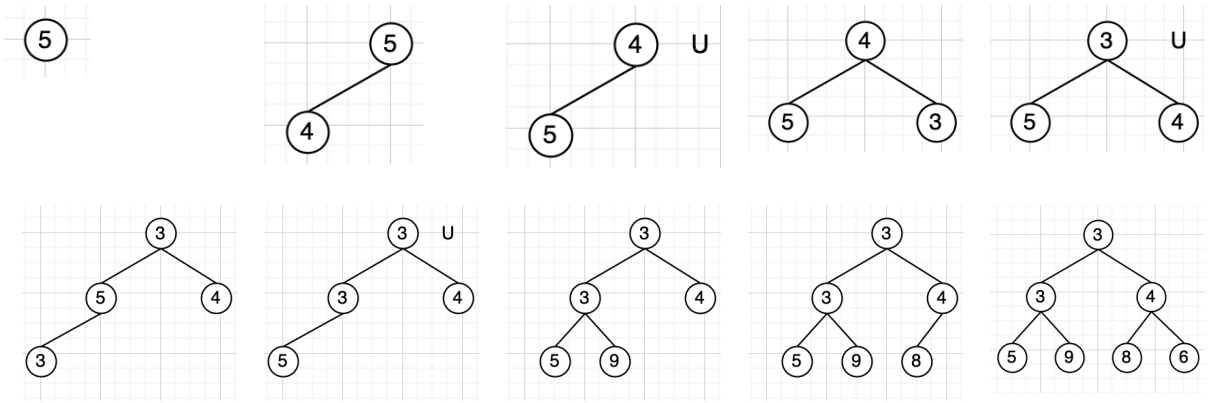
   O(h), donde h es la altura del árbol

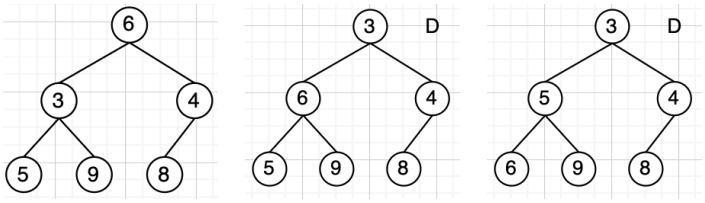
5.

Se tiene una cola con prioridad vacía implementada mediante un montículo sobre la que se llevan a cabo las siguientes operaciones:

- Se insertan de forma consecutiva las siguientes entradas (sólo mostramos las claves de tipo `Integer`, obviaremos los elementos): 5, 4, 3, 3, 9, 8, 6.
- Inmediatamente después se borra la entrada con clave mínima.

La cola con prioridad utiliza internamente el comparador estándar de enteros. **Se pide:** Dibujar los árboles (casi)completos que conforman el montículo en cada paso (añadir la entrada al árbol, «up-heap», «down-heap», borrar la entrada del árbol), etiquetando los árboles de los pasos de «up-heap» con la letra U y los árboles de los pasos de «down-heap» con la letra D.





6.

Se pide: Implementar en Java el método:

```
static <E> Map<E,Integer> contarInstancias (PositionList<E> input)
```

que recibe como parámetro una lista `input` y devuelve un `Map` cuyas claves deben ser los elementos de `input` y donde el valor asociado a cada clave debe ser el número de veces que dicha clave aparece en `input`. La lista nunca será `null`, pero podrá contener elementos `null`, que no deben ser contabilizados ni incluidos en el `Map`.

Por ejemplo, para una lista `i1 = ["a","b",null,"c","b"]`, la llamada `contarInstancias(i1)`, devolverá el map con entries `[<"a",1>,<"b",2>,<"c",1>]`; para `i2 = [4,3,null,2,4,2,null]`, la llamada `contarInstancias(i2)`, devolverá el map con entries `[<4,2>,<3,1>,<2,2>]`. Se dispone de la clase `HashTableMap`, que implementa el interfaz `Map` y que dispone de un constructor sin parámetros para crear un `Map` vacío.

```
static<E> Map<E,Integer> contarInstancias (PositionList<E> input) {
  Map<E,Integer> mapa = new HashTableMap<E,Integer>();

  Integer numVeces;
  E e;
  Position<E> cursor = input.first();
  while(cursor!=null) {
    e = cursor.element();
    if(e!=null) {
      numVeces = map.get(e);
      if(numVeces==null)
        mapa.put(e,1);
      else
        mapa.put(e,numVeces+1);
    }
    cursor = input.next(cursor);
  }

  return mapa;
}
```

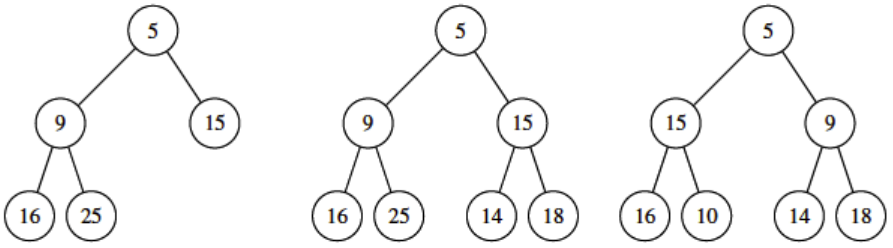
7.

Se pide: Implementar en Java el método:

```
static boolean estanHijosOrdenados (BinaryTree<Integer> tree)
```

que recibe como parámetro un árbol binario propio `tree` que no contendrá elementos `null`. Un árbol binario es *propio* si todos los nodos del árbol, o bien son nodos *hoja*, o bien tienen dos hijos. El método `estanHijosOrdenados()` debe devolver `true` si para todos los nodos con hijos del árbol, el valor de su hijo izquierdo es menor o igual que el valor de su hijo derecho, y `false` en otro caso. El árbol `tree` podría ser `null`, en cuyo caso el método debe lanzar la `IllegalArgumentException`. En caso de que el árbol `tree` esté vacío, el método debe devolver `true`.

Dados los siguientes árboles, el resultado sería:



`true` `true` `false`

```
static boolean estanHijosOrdenados(BinaryTree<Integer> tree) {
  if(tree == null)
    throw new IllegalArgumentException();

  if(tree.isEmpty())
    return true;

  return estanHijosOrdenadosRec(tree,tree.root());
}

static boolean estanHijosOrdenadosRec(BinaryTree<Integer> t,
 Position<Integer> p) {

 if(tree.isExternal(p))  // caso base
   return true;

 if(tree.letf(p) > tree.right(p)) // case base
   return false;

 return estanHijosOrdenadosRec(t, tree.letf(p)) &&
      estanHijosOrdenadosRec(t, tree.right(p));
}
```

8.

Se pide: Implementar en Java el método:

```
static <E> void imprimirPorOrdenApariciones (Map<E,Integer> map)
```

que recibe como parámetro el `Map` devuelto en el ejercicio anterior y que imprime todos los elementos contenidos en `map` en orden ascendiente respecto al número de apariciones. El argumento `map` nunca será `null`. Se dispone de la clase `HeapPriorityQueue` que implementa el interfaz `PriorityQueue` y que dispone de un constructor sin parámetros para crear una `PriorityQueue` vacía.

Por ejemplo, dado `map1 = [<"a",1>,<"b",2>,<"c",1>]` la invocación `imprimirPorOrdenApariciones(map1)` deberá imprimir `a c b` o bien `c a b`, y dado `map2 = [<4,2>,<3,1>,<2,2>]` la invocación `imprimirPorOrdenApariciones(map2)` deberá imprimir `3 2 4` o `3 4 2`.

```
static<E> void imprimirPorOrdenApariciones(Map<E,Integer> map) {
  PriorityQueue<Integer, E> cola = new HeapPriorityQueue<Integer,E>();
  Iterator<Entry<E,Integer>> it = map.entries().iterator();
  Entry<E,Integer> entry;
  while(it.hasNext()) {
    entry = it.next();
    cola.enqueue(entry.getValue(), entry.getKey());
  }

  while(!cola.isEmpty())
    System.out.print(cola.dequeue().getValue() + " ");
}
```

9.

Se pide: Implementar en Java el método

```
public boolean hasSibling(BinaryTree<E> tree, Position<E> p)
```

que indica si un nodo `p` del árbol binario `tree` tiene un nodo hermano. Puede asumirse que `tree` y `p` no son `null` y que `p` es un nodo válido del árbol. El interfaz `Tree<E>` está disponible en el Apéndice A.1. El interfaz `BinaryTree<E>` está disponible en el Apéndice A.2.

```
public boolean hasSibling(BinaryTree<E> tree, Position<E> p) {
  boolean tieneHermano = false;

  if(!tree.isRoot(p)) {
    Position<E> padre = tree.parent(p);
    tieneHermano = tree.hasLeft(padre) && tree.hasRight(padre);
  }

  return tieneHermano;
}
```

10.

Se pide: Implementar en Java el método:

```
public static <K,V> PositionList<Entry<K,V>> minList(PriorityQueue<Entry<K,V>> pq)
```

que toma como parámetro un objeto cola con prioridad. Si la cola es null o vacía entonces el método devuelve una nueva lista vacía. En otro caso el método devuelve como resultado una nueva lista con todas las entradas de la cola que tienen la clave mínima, colocándolas en la lista en el mismo orden de izquierda a derecha en el que se sacan de la cola. Por ejemplo, si se usan como claves números enteros (K es `Integer`) y como valores caracteres (V es `Character`), suponiendo que la cola tiene las entradas `(5,'X')`, `(5,'Y')`, `(5,'Z')`, `(8,'X')`, `(8,'Z')`, `(10,'Y')` entonces el método devuelve una nueva lista con las entradas `(5',X')`, `(5,'Y')`, `(5,'Z')` en ese orden. El método puede modificar la cola `pq`.

```
public static<K,V> PositionList<Entry<K,V>>
 minList(PriorityQueue<Entry<K,V>> pq) {

 PositionList<Entry<K,V>> lista = new
NodePositionList<Entry<K,V>>();

 if(pq != null && !pq.isEmpty()) {
   K key = pq.first().getKey();        // me quedo con la primera
clave
   boolean coincide = true;       // (la clave mínima)
   Entry<K,V> entry;
   while(!pq.isEmpty() && coincide) {
     entry = pq.dequeue(); // desencolo el primer elemento de la
cola
     coincide = key.equals(entry.getKey());
     if(coincide)
       lista.addLast(entry);
   }
 }

 return lista;
}
```

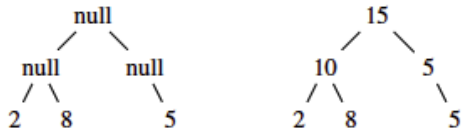
11.

Se pide: Implementar en Java **usando un método auxiliar recursivo** el método:

```
public static void sumTree(BinaryTree<Integer> tree)
```

que toma como parámetro un árbol binario `tree`. Si el árbol es null o vacío entonces el método no tiene efecto. En otro caso los elementos en los nodos hoja del árbol serán números enteros y los elementos en los nodos internos (en caso de haberlos) serán `null`. El método debe modificar el árbol para que cada nodo interno almacene como elemento el resultado de sumar el elemento en el nodo hijo izquierdo y el elemento en el nodo hijo derecho, después de que éstos hayan sido recursivamente tratados. Si un nodo interno tiene un único hijo entonces se almacena en el nodo interno el elemento del hijo después de ser tratado recursivamente.

Por ejemplo, el método `sumTree` debe modificar el árbol binario a la izquierda en la siguiente figura dejándolo como el mostrado a la derecha.




```
public static void sumTree(BinaryTree<Integer> tree) {
  if(tree!=null && !tree.isEmpty())
    sumaTreeRec(tree,tree.root());
}

public static void sumTreeRec(BinaryTree<Integer> tree,
 Position<Integer> p) {

 if(tree.isInternal(p)) {
   Integer izq = 0, der = 0;
   if(tree.hasLeft(p)) {
     sumTreeRec(tree,tree.left(p));
     izq = tree.left(p).element();
   }
   if(tree.hasRight(p)) {
     sumTreeRec(tree,tree.right(p));
     der = tree.right(p).element();
   }
   tree.set(p,izq+der);
 }

}
```

12.

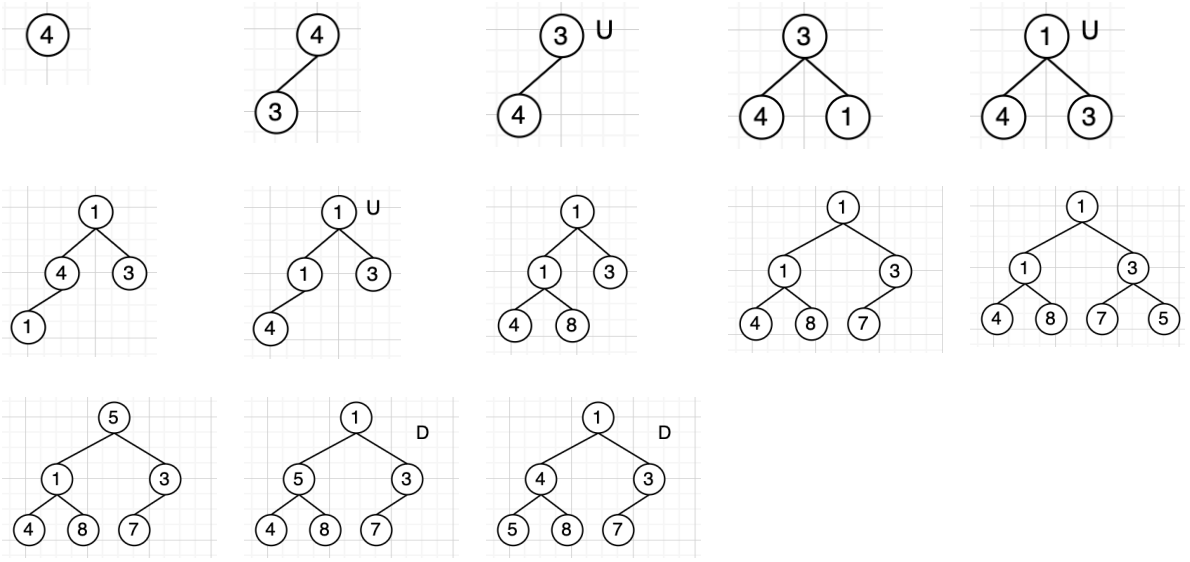
En una cola con prioridad implementada mediante un montículo se insertan de forma consecutiva las siguientes entradas (sólo mostramos las claves de tipo `Integer`, podemos ignorar los elementos): 4, 3, 1, 1, 8, 7, 5, e inmediatamente después se borra la entrada con clave mínima.

La cola con prioridad utiliza internamente un comparador estándar para `Integer` de forma que una clave k_1 tiene mayor o igual prioridad que k_2 cuando $k_1 \leq k_2$. El interfaz `PriorityQueue<K,V>` está disponible en el Apéndice A.3.

Se pide:

Dibujar el árbol (casi)completo de cada paso (añadir la entrada al árbol, borrar la entrada del árbol, «up-heap» y «down-heap») etiquetando los árboles de los pasos de «up-heap» con la letra **U** y los árboles de los pasos de «down-heap» con la letra **D**.

Dibujar el vector resultante del árbol final (tras el borrado de la entrada con clave mínima).



13.

Se ha implementado el siguiente método cuya funcionalidad se desconoce:

```
public Position<E> desconocido(BinaryTree<E> tree, Position<E> p)
{
  boolean done = false;
  while (!done) {
    if (tree.hasLeft(p))
      p = tree.left(p);
    else if (tree.hasRight(p))
           p = tree.right(p);
         else
           done = true;
  }
  return p;
}
```

El método `desconocido` recibe como parámetro un nodo `p` del árbol `tree`, y devuelve como resultado un nodo del árbol. **Se pide:** Indicar la propiedad o característica del nodo devuelto por el método, sabiendo que el método siempre es invocado pasando en `p` la raíz del árbol `tree`.

El método retorna el nodo hoja situado más a la izquierda posible.