No es necesario que entregue ninguna de las hojas del presente enunciado de examen.

P1. **Pregunta sobre la práctica**. Se desea añadir un nuevo método:

```
public void decFreqQuery (String q);
```

al interfaz `QueryDepotIF`, cuyo cometido es decrementar en uno la frecuencia de una query que ya esté en el depósito de consultas, teniendo en cuenta que se deberá eliminar la query del depósito si la frecuencia llegase a ser 0.

a) (1 Punto) Describa (no es necesario programar) cuál sería el funcionamiento del método para la representación mediante listas de queries ordenadas alfabéticamente.

b) (1 Punto) Describa (no es necesario programar) cuál sería el funcionamiento del método para la representación mediante un árbol de caracteres.

1. Se dispone de dos programas P_1 y P_2 que resuelven el mismo problema. El tiempo de ejecución de cada uno de ellos en función del tamaño del problema n (respectivamente $T_1(n)$ y $T_2(n)$) viene dado por las siguientes ecuaciones:

$$T_1(n)=\begin{Bmatrix} k_1 & si\ n\leq 10000 \\ k_2\cdot n & si\ n>10000 \end{Bmatrix}$$

$$T_2(n)=k_3+k_4\cdot n$$

que nos devuelven las unidades de tiempo que emplea cada programa en ejecutarse en función de n. Asumimos que los valores k_1, k_2, k_3 y k_4 son constantes y conocidos.

a) (1 Punto) ¿Cuál es el coste asintótico temporal en el caso peor de ambos programas? Según el análisis, ¿cuál es asintóticamente más eficiente?

b) (1 Punto) ¿Cuál de los dos programas tarda menos en ejecutarse según el valor de n? ¿Cómo podrían combinarse ambos programas para minimizar el tiempo de ejecución para cualquier valor de n?

2. Se desea disponer de un método:

```
public void removeNonMultiplesQ(int k, QueueIF<Integer> q);
```

que modifique la cola de entrada `q` eliminando de ella todos los elementos que no sean divisibles por `k`, pero manteniendo el orden de los mismos.

a) (1 Punto) Programe en Java el método `removeNonMultiplesQ` descrito anteriormente.

b) (1 Punto) Programe en Java el método:

```
public void removeNonMultiplesS(int k, StackIF<Integer> s);
```

que realice la misma operación sobre una pila de enteros.

3. Una cola con prioridad es un Tipo Abstracto de Datos similar a una cola, pero en el que los elementos tienen, además, una prioridad asignada. En una cola de este tipo, un elemento con mayor prioridad es siempre desencolado antes que otro de menor prioridad y cuando dos elementos tienen la misma prioridad, se desencolarán según su orden de entrada en la cola.

   Consideremos el caso en el que sabemos que existe un número finito y conocido k de prioridades posibles para los elementos (por ejemplo, para acceder a un avión suele haber sólo dos prioridades: embarque normal o prioritario) y las siguientes tres representaciones posibles:

   CP1) Utilizar una lista no ordenada para almacenar los elementos en el orden en el que van llegando a la cola.

   CP2) Utilizar una lista similar a la anterior, pero ordenada según la prioridad de los elementos (y el orden en el que se añaden en caso de igualdad de prioridades), de forma que un elemento que deba ser desencolado antes que otro deberá estar situado en un índice inferior en la lista.

   CP3) Agrupar los elementos en k colas, de forma que cada una de ellas almacene elementos con igual prioridad.

   a) (1.5 Puntos) Razone justificadamente el coste asintótico temporal en el caso peor de la operación de añadir un nuevo elemento a la cola de prioridad en cada una de las tres representaciones. Preste especial atención a los factores que pueden intervenir en el tamaño del problema.

   b) (1.5 Puntos) Razone justificadamente el coste asintótico temporal en el caso peor de la operación de eliminar el elemento de mayor prioridad en cada una de las tres representaciones.

   c) (0.5 Puntos) En cuanto al espacio necesario en memoria, ¿es el mismo en las tres representaciones?

   d) (0.5 Puntos) Si sabemos que el número de niveles de prioridad es mucho menor que el número de elementos esperado, ¿cuál sería la mejor representación? ¿Y en el caso contrario?

CollectionIF (Colección)

```
public interface CollectionIF<E> {
    /* Devuelve el número de elementos de la colección.        */
    public int size ();
    /* Devuelve true sii la colección no contiene elementos.   */
    public boolean isEmpty ();
    /* Devuelve true sii e está en la colección.               */
    public boolean contains (E e);
    /* Elimina todos los elementos de la colección.            */
    public void clear ();
}
```

ContainerIF (Contenedor)

```
public interface ContainerIF<E> extends CollectionIF<E> {
    /* Añade un elemento al contenedor                         */
    public void add (E e);
    /* Elimina un elemento e del contenedor                    *
     * @pre:  this.contains(e)                                 *
     * @post: !this.contains(e)                                */
    public void remove (E e);
    /* Devuelve un iterador para el contenedor                 */
    public IteratorIF<E> iterator ();
}
```

SetIF (Conjunto)

```
public interface SetIF<E> extends ContainerIF<E> {
    /* Realiza la unión del conjunto llamante con el parámetro */
    public void union (SetIF<E> s);
    /* Realiza la intersección del conjunto llamante con el    *
     * parámetro                                               */
    public void intersection (SetIF<E> s);
    /* Realiza la diferencia del conjunto llamante con el      *
     * parámetro (los elementos que están en el llamante pero  *
     * no en el parámetro)                                     */
    public void difference (SetIF<E> s);
    /* Devuelve true sii el conjunto parámetro es subconjunto  *
     * del llamante                                            */
    public boolean isSubset (SetIF<E> s);
}
```

MultiSetIF (Multiconjunto)

```
public interface MultiSetIF<E> extends ContainerIF<E> {
    /* Añade varias instancias de un elemento al multiconjunto  *
     * @pre:  n > 0 && premult = multiplicity(e)                *
     * @post: multiplicity(e) = premult + n                     */
    public void addMultiple (E e, int n);
                                    ...
```

```
    /* Elimina varias instancias de un elemento del                    *
     * multiconjunto                                                   *
     * @pre: 0<n<= multiplicity(e) && premult = multiplicity(e)        *
     * @post: multiplicity(e) = premult - n                            */
    public void removeMultiple (E e, int n);
    /* Devuelve la multiplicidad de un elemento dentro del             *
     * multiconjunto.                                                  *
     * @return: multiplicidad de e (0 si no está contenido)            */
    public int multiplicity (E e);
    /* Realiza la unión del multiconjunto llamante con el              *
     * parámetro                                                       */
    public void union (MultiSetIF<E> s);
    /* Realiza la intersección del multiconjunto llamante con          *
     * el parámetro                                                    */
    public void intersection (MultiSetIF<E> s);
    /* Realiza la diferencia del multiconjunto llamante con el         *
     * parámetro (los elementos que están en el llamante pero          *
     * no en el parámetro                                              */
    public void difference (MultiSetIF<E> s);
    /* Devuelve cierto sii el parámetro es un submulticonjunto         *
     * del llamante                                                    */
    public boolean isSubMultiSet (MultiSetIF<E> s);
}
```

SequenceIF (Secuencia)

```
/* Representa una secuencia, que es una colección de elementos      *
 * que se organizan linealmente.                                     */
public interface SequenceIF<E> extends CollectionIF<E> {
    /* Devuelve el iterador sobre la secuencia. No necesita        *
     * parámetros puesto que el recorrido es lineal y único.       */
    public IteratorIF<E> iterator ();
}
```

ListIF (Lista)

```
public interface ListIF<E> extends SequenceIF<E> {
    /* Devuelve el elemento de la lista que ocupa la posición      *
     * indicada por el parámetro.                                  *
     * @param pos la posición comenzando en 1.                     *
     * @Pre: 1 <= pos <= size().                                   *
     * @return el elemento en la posición pos.                     */
    public E get (int pos);
    /* Modifica la posición dada por el parámetro pos para que     *
     * contenga el valor dado por el parámetro e.                  *
     * @param pos la posición cuyo valor se debe modificar,        *
     *  comenzando en 1.                                           *
     * @param e el valor que debe adoptar la posición pos.         *
     * @Pre: 1 <= pos <= size().                                   */
    public void set (int pos, E e);
                                    ...
```

```
    public void insert (E elem, int pos);
    /* Elimina el elemento que ocupa la posición del parámetro  *
     * @param pos la posición que ocupa el elemento a eliminar, *
     *  comenzando en 1                                         *
     * @Pre: 1 <= pos <= size()                                 */
    public void remove (int pos);
}
```

StackIF (Pila)

```
public interface StackIF <E> extends SequenceIF<E>{
    /* Devuelve el elemento situado en la cima de la pila       *
     * @Pre !isEmpty ();                                        *
     * @return la cima de la pila                               */
    public E getTop ();
    /* Incluye un elemento en la cima de la pila. Modifica el   *
     * tamaño de la misma.                                      *
     * @param elem el elemento que se quiere añadir en la cima  */
    public void push (E elem);
    /* Elimina la cima de la pila. Modifica el tamaño de la     *
     * pila.                                                    *
     * @Pre !isEmpty ();                                        */
    public void pop ();
}
```

QueueIF (Cola)

```
public interface QueueIF<E> extends SequenceIF<E> {
    /* Devuelve el primer elemento de la cola.                  *
    /* @Pre !isEmpty()                                          *
     * @return la cabeza de la cola (su primer elemento).       */
    public E getFirst ();
    /* Incluye un elemento al final de la cola. Modifica el     *
     * tamaño de la misma.                                      *
     * @param elem el elemento que debe encolar (añadir).       */
    public void enqueue (E elem);
    /* Elimina el primer elemento de la cola. Modifica la       *
     * tamaño de la misma.                                      *
     * @Pre !isEmpty();                                         */
    public void dequeue ();
}
```

TreeIF (Árboles)

```
public interface TreeIF<E> extends CollectionIF<E> {
    /* Obtiene el elemento situado en la raíz del árbol         *
     * @Pre: !isEmpty ();                                       *
     * @return el elemento que ocupa la raíz del árbol.         */
    public E getRoot ();
                                    …
```

```
    /* Decide si el árbol es una hoja (no tiene hijos)           *
     * @return true sii el árbol es no vacío y no tiene hijos   */
    public boolean isLeaf();
    /* Devuelve el número de hijos del árbol                     */
    public int getNumChildren ();
    /* Devuelve el fan-out del árbol: el número máximo de hijos *
     * que tiene cualquier nodo del árbol                        */
    public int getFanOut ();
    /* Devuelve la altura del árbol: la distancia máxima desde  *
     * la raíz a cualquiera de sus hojas                         */
    public int getHeight ();
    /* Obtiene un iterador para el árbol.                        *
     * @param mode el tipo de recorrido indicado por los         *
     * valores enumerados definidos en cada TAD concreto.        */
    public IteratorIF<E> iterator (Object mode);
}
```

GTreeIF (Árbol General)

```
public interface GTreeIF<E> extends TreeIF<E> {
    /* Valor enumerado que indica los tipos de recorridos       *
     * ofrecidos por los árboles generales.                      */
    public enum IteratorModes {
        PREORDER, POSTORDER, BREADTH
    }
    /* Modifica la raíz del árbol.                               *
     * @param el elemento que se quiere poner como raíz del      *
     * árbol.                                                    */
    public void setRoot (E e);
    /* Obtiene los hijos del árbol llamante.                     *
     * @return la lista de hijos del árbol (en el orden en que  *
     * estén almacenados en el mismo.                            */
    public ListIF <GTreeIF<E>> getChildren ();
    /* Obtiene el hijo que ocupa la posición dada por parámetro.*
     * @param pos la posición del hijo que se desea obtener,     *
     *  comenzando en 1.                                         *
     * @Pre 1 <= pos <= getChildren ().size ();                  *
     * @return el árbol hijo que ocupa la posición pos.          */
    public GTreeIF<E> getChild (int pos);
    /* Inserta un árbol como hijo en la posición pos.            *
     * @param pos la posición que ocupará el árbol entre sus     *
     * hermanos, comenzando en 1.                                *
     * Si pos == getChildren ().size () + 1, se añade como       *
     * último hijo.                                              *
     * @param e el hijo que se desea insertar.                   *
     * @Pre 1<= pos <= getChildren ().size () + 1                */
    public void addChild (int pos, GTreeIF<E> e);
    /* Elimina el hijo que ocupa la posición parámetro.          *
     * @param pos la posición del hijo con base 1.               *
     * @Pre 1 <= pos <= getChildren ().size ();                  */
    public void removeChild (int pos);
}
```

BTreeIF (Arbol Binario)

```
public interface BTreeIF<E> extends TreeIF<E>{
    /* Valor enumerado que indica los tipos de recorrido        *
     * ofrecidos por los árboles de binarios.                   */
    public enum IteratorModes {
        PREORDER, POSTORDER, BREADTH, INORDER, RLBREADTH
    }
    /* Obtiene el hijo izquierdo del árbol llamante.            *
     * @return el hijo izquierdo del árbol llamante.            */
    public BTreeIF<E> getLeftChild ();
    /* Obtiene el hijo derecho del árbol llamante.              *
     * @return el hijo derecho del árbol llamante.              */
    public BTreeIF<E> getRightChild ();
    /* Modifica la raíz del árbol.                              *
     * @param el elemento que se quiere poner como raíz del     *
     * árbol.                                                   */
    public void setRoot (E e);
    /* Pone el árbol parámetro como hijo izquierdo del árbol    *
     * llamante. Si ya había hijo izquierdo, el antiguo dejará  *
     * de ser accesible (se pierde).                            *
     * @Pre: !isEmpty()                                         *
     * @param child el árbol que se debe poner como hijo        *
     *        izquierdo.                                        */
    public void setLeftChild (BTreeIF <E> child);
    /* Pone el árbol parámetro como hijo derecho del árbol      *
     * llamante. Si ya había hijo izquierdo, el antiguo dejará  *
     * de ser accesible (se pierde).                            *
     * @Pre: !isEmpty()                                         *
     * @param child el árbol que se debe poner como hijo        *
     *        derecho.                                          */
    public void setRightChild (BTreeIF <E> child);
    /* Elimina el hijo izquierdo del árbol.                     */
    public void removeLeftChild ();
    /* Elimina el hijo derecho del árbol.                       */
    public void removeRightChild ();
}
```

BSTreeIF (Árbol de Búsqueda Binaria)

```
public interface BSTreeIF<E extends Comparable<E>> extends TreeIF<E> {
    /* Valor enumerado que indica los tipos de recorrido        *
     * ofrecidos por los árboles de búsqueda binaria.           */
    public enum IteratorModes { DIRECTORDER, REVERSEORDER }
    /* Valor enumerado que indica cuál es la ordenación de los  *
     * elementos dentro del árbol (ascendente o descendente).   */
    public enum Order { ASCENDING, DESCENDING }
    /* Añade un elemento no contenido previamente en el árbol   *
     * @Pre: !contains(e)                                       *
     * @Post: contains(e)                                       */
    public void add(E e);
                                    …
```

```
    /* Elimina un elemento previamente contenido en el árbol     *
     * @Pre: contains(e)                                         *
     * @Post: !contains(e)                                       */
    public void remove(E e);
    /* Devuelve en qué orden están almacenados los elementos     */
    public Order getOrder();
}
```

IteratorIF (Iterador)

```
/* Representa un iterador sobre un Tipo Abstracto de Datos.       */
public interface IteratorIF<T>{
    /* Obtiene el siguiente elemento de la iteración.             *
     * @Pre: hasNext ();                                          *
     * @return el siguiente elemento de la iteración,             */
    public T getNext ();
    /* Comprueba si aún quedan elementos por iterar.              *
     * @return true sii el iterador dispone de más elementos.     */
    public boolean hasNext ();
    /* Vuelve la posición del iterador al principio. Esto         *
     * permite reutilizar un iterador evitando crear otro nuevo.  */
    public void reset ();
}
```