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

P1. **Pregunta sobre la práctica**. Supongamos que se desea almacenar la información del número de veces que una canción ha sido reproducida. En base a esto:

a) (0.75 Puntos) Indique, de manera justificada, en qué clase de las que ha implementado en la práctica debería realizar cambios en la estructura de datos (y cuáles serían) para poder almacenar esta información.

b) (0.5 Puntos) ¿Sería necesario añadir algún método público a alguna de las clases ya implementadas? Es decir, ¿sería necesario modificar alguna de las interfaces que se han dado? Justifique su respuesta.

c) (0.75 Puntos) Describa (no es necesario implementar) cómo se actualizaría esta información indicando qué métodos ya implementados necesitaría modificar y qué cambios serían necesarios. Justifique su respuesta.

1. Tenemos un conjunto A en el que cada elemento es, a su vez, un conjunto de enteros. Describa (no es necesario implementar) cómo realizaría las siguientes operaciones sobre dicho conjunto A (**nota**: se valorará la eficiencia):

   a) (0.75 Puntos) Se desea obtener el conjunto "alfabeto de A" definido como el conjunto de todos los enteros distintos presentes en al menos uno de los conjuntos de A.

   b) (0.75 Puntos) Se desea obtener el conjunto "núcleo de A" definido como el conjunto de todos los enteros distintos que aparezcan en todos y cada uno de los conjuntos de A.

   c) (0.5 Puntos) ¿Cuál sería el coste asintótico temporal en el caso peor de los algoritmos que ha descrito en los apartados anteriores? Justifique su respuesta.

2. (2 Puntos) Se desea un método:

   ```
   StackIF<T> palindromeStack()
   ```

   dentro del interfaz `StackIF<T>` que, dada una pila llamante s con n elementos devuelva una pila con 2n-1 elementos, tal que los n primeros elementos apilados coincidan con los de s (y en el mismo orden) y que, además, sea capicúa. Por ejemplo:

   ```
   s = [5,3,1,8,5,2] ← (cima)
   s.palindromeStack() = [5,3,1,8,5,2,5,8,1,3,5] ← (cima)
   ```

3. Queremos un TAD que nos permita representar matrices bidimensionales, pero con la particularidad de que **no podemos utilizar arrays como estructura de datos** para representar los elementos del TAD.

   Las operaciones que, como mínimo, debe ofrecer el TAD son las siguientes:

   O1. Una operación que nos permita obtener el elemento (i,j) de la matriz (es decir, el situado en la fila i, columna j).

   O2. Una operación que nos permita modificar el elemento (i,j) de la matriz, substituyendo el contenido actual de dicha posición por un elemento e.

O3. Una operación que nos permita obtener la fila i-ésima de la matriz como una lista de elementos.

O4. Una operación que nos permita obtener la columna j-ésima de la matriz como una lista de elementos.

En una primera aproximación escogemos una representación en forma de lista de filas, de forma que el primer elemento de esa lista será la fila 1, el segundo la fila 2 y así sucesivamente hasta la fila F. Cada fila i será a su vez una lista de los elementos presentes en dicha fila de la matriz: el primer elemento será el presente en la posición (i,1), el segundo será el presente en la posición (i,2) y así sucesivamente hasta el elemento (i,C), que será el último de la fila.

Se pide lo siguiente (no es necesario programar):

a) (1 Punto) Describa (sin limitarse a copiar el enunciado) el funcionamiento de las operaciones O1 a O4 según esta representación. Si necesita alguna otra operación adicional en la que apoyarse, su funcionamiento también deberá ser descrito. Se valorará que el funcionamiento de estas operaciones no sea artificialmente costoso con respecto a la representación dada.

b) (1 Punto) Analice el coste asintótico temporal en el caso peor de las operaciones O1 a O4 según el funcionamiento descrito en el apartado anterior.

c) (0.5 Puntos) Si ahora tenemos en cuenta que las operaciones O3 y O4 son las más frecuentemente utilizadas, ¿considera adecuada esta representación para los elementos del TAD? ¿Por qué?

d) (1.5 Puntos) Proponga una nueva representación para los elementos del TAD que tenga en cuenta el hecho de que O3 y O4 son las operaciones más utilizadas (puede acompañar a su respuesta una representación gráfica de la estructura si considera que esto puede ayudar a su comprensión). Calcule el coste de todas las operaciones en esta nueva representación e indique qué cambios habría que realizar en el funcionamiento de las operaciones bajo esta representación.

Justifique todas sus respuestas.

CollectionIF (Coleccion)

```
/* Representa una colección de elementos. Una colección no        *
 * tiene orden.                                                   */
public interface CollectionIF<E> {
    /* Devuelve el número de elementos de la colección.           *
     * @return: cardinalidad de la colección.                     */
    public int size ();
    /* Determina si la colección está vacía.                      *
     * @return: size () == 0                                      */
    public boolean isEmpty ();
    /* Determina la pertenencia del parámetro a la colección      *
     * @param: el elemento cuya pertenencia se comprueba.         *
     * @return: param \in self                                    */
    public boolean contains (E e);
    /* Elimina todos los elementos de la colección.               */
    public void clear ();
    /* Devuelve un iterador sobre la colección.                   *
     * @return: un objeto iterador para los elementos de          *
     * la colección.                                              */
    public IteratorIF<E> iterator ();
}
```

SetIF (Conjunto)

```
/* Representa un conjunto de elementos. Se trata del concepto     *
 * matemático de conjunto finito (no tiene orden).                */
public interface SetIF<E> extends CollectionIF<E> {
    /* Devuelve la unión del conjunto llamante con el parámetro   *
     * @param: el conjunto con el que realizar la unión           *
     * @return: self \cup @param                                  */
    public SetIF<E> union (SetIF<E> s);
    /* Devuelve la intersección con el parámetro.                 *
     * @param: el conjunto con el que realizar la intersección.   *
     * @return: self \cap @param                                  */
    public SetIF<E> intersection (SetIF<E> s);
    /* Devuelve la diferencia con el parámetro (los elementos     *
     * que están en el llamante pero no en el parámetro).         *
     * @param: el conjunto con el que realizar la diferencia.     *
     * @return: self \setminus @param                             */
    public SetIF<E> difference (SetIF<E> s);
    /* Determina si el parámetro es un subconjunto del llamante.  *
     * @param: el posible subconjunto del llamante.               *
     * @return: self \subseteq @param                             */
    public boolean isSubset (SetIF<E> s);
}
```

ListIF (Lista)

```
/* Representa una lista de elementos.                                    */
public interface ListIF<E> extends CollectionIF<E>{
    /* Devuelve el elemento de la lista que ocupa la posición            *
     * indicada por el parámetro.                                        *
     * @param: pos la posición comenzando en 1.                          *
     * @Pre: 1 \leq pos \leq size()                                      *
     * @return: el elemento en la posición pos.                          */
    public E get (int pos);
    /* Modifica la posición dada por el parámetro pos para que           *
     * conten: pos la posición cuyo valor se debe modificar,             *
     * comenzando en 1.                                                  *
     * @param: e el valor que debe adoptar la posición pos.              *
     * @Pre: 1 \leq pos \leq size()                                      */
    public void set (int pos, E e);
    /* Inserta un elemento en la Lista.                                  *
     * @param: elem El elemento que hay que añadir.                      *
     * @param: pos La posición en la que se debe añadir elem,            *
     * comenzando en 1.                                                  *
     * @Pre: 1 \leq pos \leq size()+1                                    */
    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 \leq pos \leq size()                                      */
    public void remove (int pos);
}
```

StackIF (Pila)

```
/* Representa una pila de elementos.                                     */
public interface StackIF <E> extends CollectionIF<E>{
    /* Obtiene el elemento 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)

```
/* Representa una cola de elementos.                                    */
public interface QueueIF <E> extends CollectionIF<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 (crece en una unidad).                        *
     * @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 (decrece en una unidad).                      *
     * @Pre !isEmpty();                                                 */
    public void dequeue ();
}
```

TreeIF (Arbol general)

```
/* Representa un árbol n-ario de elementos, donde el número de         *
 * hijos de un determinado nodo no está determinado de antemano        *
 * (fan-out no prefijado, no necesariamente igual en cada nodo).       */
public interface TreeIF<E> extends CollectionIF<E>{
    public int PREORDER = 0;
    public int POSTORDER = 1;
    public int BREADTH = 2;
    /* Obtiene la raíz del árbol (único elemento sin antecesor).        *
     * @Pre: !isEmpty ();                                               *
     * @return el elemento que ocupa la raíz del árbol.                 */
    public E getRoot ();
    /* 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.                            *
     * @Pre: !isEmpty ();                                               *
     * @return la lista de hijos del árbol (en el orden en que         *
     * están almacenados en el mismo).                                 */
    public ListIF <TreeIF <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 \leq pos \leq getChildren().size() && !isEmpty()          *
     * @return el árbol hijo que ocupa la posición pos.                 */
    public TreeIF<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 \leq pos \leq getChildren().size()+1 && !isEmpty()      */
    public void addChild (int pos, TreeIF<E> e);
    /* Elimina el hijo que ocupa la posición parámetro.               *
     * @param pos la posición del hijo con base 1.                    *
     * @Pre 1 \leq pos \leq getChildren().size() && !isEmpty()        */
    public void removeChild (int pos);
    /* Determina si el árbol llamante es una hoja.                    *
     * @Pre: !isEmpty (); (un arbol vacio no se considera hoja)       *
     * @return el árbol es una hoja (no tiene hijos).                 */
    public boolean isLeaf ();
    /* Obtiene un iterador para el árbol.                             *
     * @param traversal el tipo de recorrido indicado por las         *
     * constantes PREORDER (preorden o profundidad), POSTORDER        *
     * (postorden) o BREADTH (anchura)                                *
     * @return un iterador según el recorrido indicado               */
    public IteratorIF<E> iterator (int traversal);
}
```

BTreeIF (Arbol Binario)

```
/* Representa un arbol binario de elementos                          */
public interface BTreeIF<E> extends CollectionIF<E>{
    public int PREORDER = 0;
    public int POSTORDER = 1;
    public int BREADTH = 2;
    public int INORDER = 3;
    public int RLBREADTH = 4;
    /* Obtiene la raíz del árbol (único elemento sin antecesor).      *
     * @Pre: !isEmpty ();                                             *
     * @return: el elemento que ocupa la raíz del árbol.              */
    public E getRoot ();
    /* Obtiene el hijo izquierdo del árbol llamante o un árbol        *
     * vacío en caso de no existir.                                   *
     * @Pre: !isEmpty ();                                             *
     * @return: un árbol, bien el hijo izquierdo bien uno vacío       *
     * de no existir tal hijo.                                        */
    public BTreeIF<E> getLeftChild ();
    /* Obtiene el hijo derecho del árbol llamante o un árbol          *
     * vacío en caso de no existir.                                   *
     * @Pre: !isEmpty ();                                             *
     * @return: un árbol, bien el hijo derecho bien uno vacío        *
     * de no existir tal hijo.                                        */
    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 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 árbol que se debe poner como hijo derecho.   */
    public void setRightChild (BTreeIF <E> child);
    /* Elimina el hijo izquierdo del árbol.                       *
     * @Pre: !isEmpty ();                                         */
    public void removeLeftChild ();
    /* Elimina el hijo derecho del árbol.                         *
     * @Pre: !isEmpty ();                                         */
    public void removeRightChild ();
    /* Determina si el árbol llamante es una hoja.                *
     * @Pre: !isEmpty (); (un arbol vacio no se considera hoja)   *
     * @return: true sii el árbol es una hoja (no tiene hijos).   */
    public boolean isLeaf ();
    /* Obtiene un iterador para el árbol.                         *
     * @param: traversal el tipo de recorrido indicado por las    *
     * constantes PREORDER (preorden o profundidad), POSTORDER    *
     * (postorden), BREADTH (anchura), INORDER (inorden) o        *
     * RLBREADTH (anchura de derecha a izquierda).                *
     * @return: un iterador según el recorrido indicado.         */
    public IteratorIF<E> iterator (int traversal);
}
```

ComparatorIF (Comparador)

```
/* Representa un comparador entre elementos respecto a una       *
 * relación de (al menos) preorden.                              */
public interface ComparatorIF<E>{
    /* Sean a, b elementos de un conjunto dado y \sqsubset la    *
     * relación que establece un preorden entre ellos (nótese    *
     * que \sqsupset sería la relación recíproca, es decir, en   *
     * sentido opuesto a \sqsubset):                             */
    public static int LT = -1; // Less than: a \sqsubset b
    public static int EQ = 0;  // Equals: !(a \sqsubset b) &&
                               //      && !(a \sqsupset b)
    public static int GT = 1;  // Greater than: a \sqsupset b
    /* Compara dos elementos respecto a un preorden e indica su  *
     * relación respecto al mismo, es decir, cuál precede al     *
     * otro mediante esa relación.                               *
     * @param a el primer elemento.                              *
     * @param b el segundo elemento.                             *
     * @return LT sii a \subsesq b;                              *
```

```
     * EQ sii !(a \subsetsq b) && !(a \sqsupset b)                    *
     * GT sii a \sqsupset b                                           */
    public int compare (E a, E b);
    /* Determina si el primer parámetro precede en el preorden       *
     * al segundo (a < b).                                           *
     * @param a el primer elemento.                                  *
     * @param b el segundo elemento.                                 *
     * @return a \sqsubset b;                                        */
    public boolean lt (E a, E b);
    /* Determina si el primer parámetro es igual al segundo en       *
     * el preorden.                                                  *
     * @param a el primer elemento.                                  *
     * @param b el segundo elemento.                                 *
     * @return a EQ b sii !(a \sqsubset b) && !(a \sqsupset b)       */
    public boolean eq (E a, E b);
    /* Determina si el primer parámetro sucede en el preorden        *
     * al segundo (a > b).                                           *
     * @param a el primer elemento.                                  *
     * @param b el segundo elemento.                                 *
     * @return a GT b sii a \sqsupset b                              */
    public boolean gt (E a, E b);
}
```

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 ();
}
```