

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

P1. Pregunta sobre la práctica. Se quiere dotar al depósito de queries de un histórico, de manera que sólo se almacenen las últimas n queries individuales introducidas (siendo n un número conocido que se le pasa por parámetro al constructor).

- a) (1 Punto) Describa (no es necesario programar) cómo puede representarse dicho histórico mediante los TADs estudiados en la asignatura. Justifique su respuesta.
- b) (1 Punto) Identifique las operaciones de `QueryDepotIF` que se verían afectadas por la inclusión del histórico y detalle cómo deberían modificarse (no es necesario programarlas).

1. Se pide:

- a) (1 Punto) ¿Cuál es el coste asintótico temporal en el caso peor de localizar un elemento cualquiera en un **árbol de búsqueda binaria**? ¿Por qué? Justifique su respuesta.
- b) (1 Punto) ¿Cómo consiguen los **árboles AVL** mejorar este coste? Justifique su respuesta.

2. (2 Puntos) Se tienen dos pilas de enteros que están ordenadas de menor a mayor (es decir, el número más pequeño está en la cima de cada pila). Se desea programar un método:

```
public StackIF<Integer> mixStacks( StackIF<Integer> s1,  
                                   StackIF<Integer> s2 );
```

que combine las pilas `s1` y `s2` en una nueva pila de enteros que contenga todos los elementos presentes en `s1` y `s2` y que esté ordenada de menor a mayor. No puede utilizar iteradores y las pilas pasadas por parámetro deben conservar su contenido a la salida del método.

3. Un **Multiconjunto** es una colección que permite almacenar elementos y en la que puede haber más de una instancia de cada uno de ellos. Al número de repeticiones de un elemento se le llama multiplicidad.

Nota: recuerde que dispone del interfaz de Multiconjunto en las hojas de interfaces de este enunciado.

Consideremos los siguientes escenarios:

MC1) Sabemos que los elementos pueden ordenarse, por lo que utilizamos una lista ordenada (crecientemente mediante el orden de los elementos) de pares `<elemento,multiplicidad>` como estructura de datos para implementar un Multiconjunto.

MC2) Sabemos que los elementos pueden ordenarse, por lo que utilizamos un árbol AVL (ordenado crecientemente mediante el orden de los elementos) de pares `<elemento,multiplicidad>` como estructura de datos para implementar un Multiconjunto.

MC3) No sabemos si los elementos pueden ordenarse o no, por lo que utilizamos una lista no ordenada de pares `<elemento,multiplicidad>` para implementar el Multiconjunto.

- a) (1.5 Puntos) Razone justificadamente el coste asintótico temporal en el caso peor del método `addMultiple(e,n)` (ver interfaz) en cada una de las tres representaciones. Preste especial atención a los factores que pueden intervenir en el tamaño del problema.
- b) (2 Puntos) Explique cómo se podría implementar el método `union(s)` (ver interfaz) en cada una de las tres representaciones, teniendo en cuenta las características de cada una de ellas. Justifique su respuesta.
- c) (0.5 Puntos) En base a las respuestas dadas en los apartados anteriores, ¿cuál considera que sería la mejor representación si los elementos pueden ordenarse? Justifique su respuesta.

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);
    ...
}
```

Interfaces de los TAD del Curso

```
/* 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);
    ...
}
```

Interfaces de los TAD del Curso

```
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 ();
```

...

Interfaces de los TAD del Curso

```
/* 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);
}
```

No olvide consignar su nombre y DNI en todas las hojas que entregue

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);
    ...
}
```

Interfaces de los TAD del Curso

```
/* 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 ();
}
```