

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 Query mostFrequentQuery ();
```

al interfaz `QueryDepotIF`, cuyo cometido es devolver, **en tiempo constante**, la consulta más frecuente (o una cualquiera de las más frecuentes a igualdad de frecuencias) que esté almacenada en ese momento en el depósito de consultas.

- a) (1 Punto) Indique los cambios necesarios en la estructura de datos y qué métodos de las clases que implementan `QueryDepotIF` se verían afectados. Describa los cambios necesarios (no es necesario programar) en dichos métodos.
- b) (1 Punto) Supongamos que también existiese un método:

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

que decrementa en uno la frecuencia de una query almacenada en el repositorio. Describa (no es necesario programar) qué acciones debería realizar este método para que el método `mostFrequentQuery` pueda realizar su tarea en tiempo constante.

- 1. Un **Árbol Ternario** es un árbol que puede almacenar hasta un máximo de tres hijos a los que podemos denominar hijo izquierdo, hijo derecho e hijo central.
 - a) (1.5 Puntos) Diseñe un interfaz `TTreeIF<E>` (Ternary Tree) que extienda el interfaz `TreeIF<E>` (que puede consultar en las hojas de interfaces) y que defina las operaciones que pueden realizarse sobre un árbol ternario.
 - b) (1 Punto) Compare el interfaz `TTreeIF<E>` realizado en el apartado anterior con el interfaz de árbol binario `BTreeIF<E>` (que está en las hojas de interfaces). Desde el punto de vista de Java, ¿sería posible definir `TTreeIF<E>` como una extensión de `BTreeIF<E>`? ¿Sería coherente hacerlo desde el punto de vista de Tipos Abstractos de Datos? Justifique sus respuestas.

- 2. Se desea programar un método:

```
public Integer predecesor( BTreeIF<Integer> bt );
```

que recibe un árbol binario `bt` y devuelve el valor entero que precede al valor contenido en la raíz en un recorrido en inorden del árbol o bien el valor `null` si el nodo raíz fuera el primer nodo del recorrido en inorden.

- a) (1.5 Puntos) Programe el método `predecesor` sin generar el recorrido en inorden del árbol.
- b) (0.5 Puntos) Calcule el coste asintótico temporal en el caso peor del método programado en el apartado anterior.

3. A diferencia del tipo lista definido por `ListIF<E>` (ver en las hojas de interfaces de este enunciado) una lista ordenada mantiene siempre sus elementos ordenados respecto a un orden de precedencia definido mediante una operación:

```
public Boolean precede(E a, E b);
```

que devuelve verdadero si y sólo si el elemento `a` precede al elemento `b` en dicho orden. Por ejemplo, si `E` es el tipo `Integer`, podría ser la operación \leq la que definiese ese orden.

- a) (1.5 Puntos) Para cada una de las operaciones de `ListIF<E>`, indíquese de forma razonada si tendría cabida o no en una interfaz `OrderedListIF<E>` de lista ordenada y, en caso positivo, cuáles serían los parámetros adecuados y si variarían su precondition y postcondition con respecto a las de `ListIF<E>`.
- b) (1 Punto) Indíquese también si alguna de las operaciones heredadas de secuencia (`SequenceIF<E>`) y de colección (`CollectionIF<E>`) podría implementarse de forma más eficiente teniendo en cuenta las propiedades de la lista ordenada.
- c) (1 Punto) Considérese el caso en el que los elementos de la lista son cadenas de caracteres que se ordenan según un orden alfabético. ¿Cuál sería la dependencia del coste temporal, en caso peor, con el tamaño de la lista y el tamaño de las cadenas de caracteres?

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 (Árbol 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 ();
}
```