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.

La clave en este ejercicio es el hecho de que el método `mostFrequentQuery` debe devolver una respuesta en tiempo constante. Para ello será necesario que el método disponga de acceso directo a la query más frecuente que estuviera almacenada en cada momento en el repositorio de consultas. Es decir, esto nos lleva a que es necesario añadir un atributo, llamémoslo `mfq` (most frequent query), de tipo `Query` que siempre contenga la query más frecuente.

Ahora bien, para que en `mfq` siempre tengamos la query más frecuente, es necesario modificar aquellos métodos que pueden alterar la frecuencia de las queries. Uno obvio es `incFreqQuery`, que incrementa en 1 la frecuencia de la query cuyo texto recibe como parámetro (y la introduce en el depósito si antes no estaba en él). Los cambios a realizar en dicho método son sencillos: una vez se aumenta la frecuencia de la query, se debe comprobar si la nueva frecuencia es mayor que la query contenida en `mfq`. En caso afirmativo, se debe modificar el valor de `mfq`, ya que ese incremento de frecuencia ha originado una query más frecuente que la que estaba almacenada con anterioridad.

Sin embargo, no debemos olvidarnos del constructor de la clase. Si bien el constructor no aparece en el interfaz (por motivos obvios), sí que es uno de los métodos de las clases que implementan el interfaz. Cuando construimos el repositorio no debemos olvidar darle un valor apropiado al atributo `mfq`. Dicho valor ya dependerá de qué es lo que haga el constructor (si construye un repositorio vacío puede darle el valor `null`, pero si construye un repositorio no vacío, deberá darle el valor adecuado, con una de las queries de entre las que tengan la frecuencia máxima).

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.

La situación es similar al apartado anterior, dado que ese método `decFreqQuery` va a decrementar en 1 la frecuencia de la query cuyo texto reciba como parámetro, puede afectar a cuál es la query más frecuente.

Lo primero que ha de hacer el método es buscar la query en el repositorio (que, según nos indican en el enunciado, está presente y eso debería reflejarse en su precondición) y decrementar en uno su frecuencia. Si fuera 0, se tendría que eliminar del repositorio.

Una vez modificada la frecuencia de la query (y eliminada si hubiera sido necesario), hay que comprobar si el texto de la query coincide con el de la query más frecuente. Si no coincide, no hay que hacer más, pero si coincide, puede resultar que al decrementar la frecuencia de la query más frecuente, dicha query ya no sea la más frecuente. Así, `mostFrequentQuery` podrá acceder correctamente a la query más frecuente en tiempo constante (a costa de que el "trabajo sucio" se lo ha hecho `decFreqQuery`).

Así pues, en caso de coincidencia hay que buscar por todo el repositorio y comprobar si existe una query con una frecuencia superior a la de esa query y, en caso afirmativo, actualizar el valor contenido en `mfq`. Pongamos un ejemplo para verlo más claro: si la query más frecuente era "casa" con frecuencia 28 y decrementamos su frecuencia, ahora tendría 27. En este caso, habría que buscar en el repositorio una query con una frecuencia de 28 (que podría existir perfectamente). En caso de encontrarla esa sería la query más frecuente (o, al menos, una de ellas, que es lo que debemos almacenar en `mfq`).

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.

   El interfaz `TTreeIF<E>` sería muy similar al interfaz `BTreeIF<E>` que también puede consultarse en las hojas de interfaces que acompañan al examen. Por ello, en esta solución vamos a limitarnos a comentar y explicar las diferencias entre ambos interfaces.

   En primer lugar, **todos** los métodos presentes en el interfaz `BTreeIF<E>` también han de estar en `TTreeIF<E>`, pues se refieren a los hijos izquierdo y derecho (que también estarían en un árbol ternario) o bien el método `setRoot` que nos permite modificar la raíz.

Si nos fijamos, en el interfaz `BTreeIF<E>` tenemos tres métodos para cada hijo: un getter, un setter y un remove, por lo que para el hijo central se van a necesitar también esos tres métodos:

```
/* Obtiene el hijo central del árbol llamante.              *
 * @return el hijo central del árbol llamante.              */
public BTreeIF<E> getCenterChild ();
/* Pone el árbol parámetro como hijo central del árbol      *
 * llamante. Si ya había hijo central, el antiguo dejará    *
 * de ser accesible (se pierde).                            *
 * @Pre: !isEmpty()                                         *
 * @param child el árbol que se debe poner como hijo        *
 *        central.                                          */
public void setCenterChild (BTreeIF <E> child);
/* Elimina el hijo central del árbol.                       */
public void removeCenterChild ();
```

Pero en los interfaces de los árboles tenemos, además, la definición de los posibles recorridos que cada tipo de árbol nos permite. Aquí sí que hay que hacer una distinción, ya que en los árboles binarios el recorrido en inorden (identificado como ***`INORDER`***) queda totalmente definido, ya que la raíz sólo puede visitarse en un punto: entre el hijo izquierdo y el derecho.

Sin embargo, en los árboles ternarios tenemos tres hijos. Un recorrido en inorden debe visitar la raíz entre las visitas a dos de sus hijos pero, ¿qué dos de entre los tres que tenemos? Sin explicaciones adicionales no se puede saber.

Así pues, incluir un recorrido en inorden sin más explicación no es adecuado para los árboles ternarios, por lo que no debería estar. En todo caso, podría ser válido considerar dos recorridos diferentes en "*inorden*", uno entre el hijo izquierdo y el central y otro entre el central y el derecho.

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.

En cuanto a la primera pregunta: sí, es posible definir `TTreeIF<E>` como una extensión de `BTreeIF<E>`, para ello habría que añadir las cabeceras de los tres métodos descritos en el apartado anterior y modificar el valor enumerado que nos define qué posibles iteradores pueden crearse para un árbol ternario.

Es más, incluso podría crearse la clase `TTree<E>` como una extensión de `BTree<E>`. Para ello habría que añadir un atributo `centerChild` para que los nuevos métodos tengan acceso a él y modificar los métodos relacionados con los recorridos, para que también tengan en cuenta el hijo central.

Sin embargo, desde el punto de vista de los Tipos Abstractos de Datos, esto no tendría sentido, ya que sería como admitir que todo árbol ternario es, a su vez, un árbol binario, lo cual, conceptualmente hablando, es incorrecto.

Se deja como ejercicio definir el interfaz `TTreeIF<E>` de dos formas diferentes: como extensión de `TreeIF<E>` y como extensión de `BTreeIF<E>`, así como una clase `TTree<E>` que implemente dicho interfaz en ambas versiones.

2. Se desea programar un método:

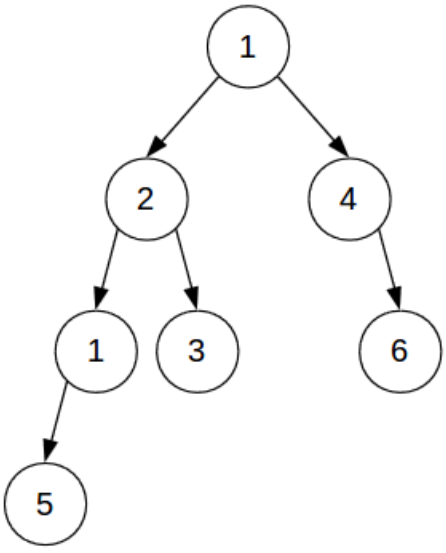
```
public Integer predecesor( BTreeIF<Integer> bt );
```

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.

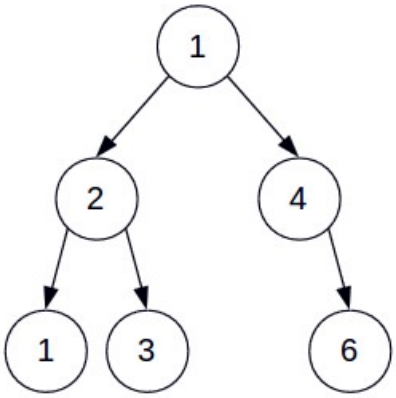
a) (1.5 Puntos) Programe el método predecesor sin generar el recorrido en inorden del árbol.

Antes de mostrar la solución propuesta y su explicación, vamos a indicar uno de los errores que más se han repetido en las respuestas a esta pregunta. En el propio enunciado se indica que se ha de hacer **sin generar el recorrido en inorden del árbol**, pero muchas respuestas comienzan generando el iterador del árbol en inorden, para luego buscar el valor contenido en la raíz y devolver el valor que le precede en la secuencia.

Aparte de incumplir lo que se está pidiendo en el enunciado ("*sin generar el recorrido en inorden*"), este método no puede utilizarse para resolver el problema, ya que al hacer un iterador estamos secuencializando el árbol y, en consecuencia, perdiendo su estructura. Consideremos los siguientes árboles:



Árbol 1: 5 – 1 – 2 – 3 – **1** – 4 – 6



Árbol 2: 1 – 2 – 3 – **1** – 4 – 6

Si secuencializamos los árboles y nos limitamos a devolver el predecesor del valor contenido en la raíz dentro la secuencia, vemos que en el primer árbol nos encontraríamos con otro nodo que contiene el mismo valor que la raíz (algo que no está prohibido por el enunciado) y se devolvería erróneamente el valor `5`. En el segundo árbol, el primer nodo de la secuencia contiene el mismo valor que la raíz del árbol y, en consecuencia, se devolvería `null`.

Sin embargo, a la vista de ambos árboles es claro ver que el resultado correcto del método predecesor ha de ser `3` para los dos, porque es el valor que precede al contenido en la raíz (resaltado en negrita). ¿Cuál es el problema? El iterador nos ha secuencializado el árbol, perdiendo así la información de cuál es el nodo raíz. No podemos distinguir si es el primer o el segundo `1` y, en consecuencia, no podemos resolver correctamente el ejercicio.

¿Cómo hemos de hacerlo? Bien, un recorrido en inorden empieza recorriendo en inorden el hijo izquierdo, luego visita la raíz y acaba recorriendo en inorden el hijo derecho. Es decir, el predecesor de la raíz en el recorrido en inorden será el último nodo que se visite dentro de su hijo izquierdo, que será el hijo derecho, del hijo derecho, del hijo derecho … del hijo derecho del hijo izquierdo de la raíz.

Así pues, la estrategia a seguir será: partimos de la raíz del árbol y vamos a su hijo izquierdo (si existe). Desde ahí vamos avanzando por los hijos derechos hasta que no haya más, momento en el que nos habremos situado en el nodo que se visita justo antes de la raíz en el recorrido en inorden, con lo que ya podremos devolver el resultado correcto.

Por lo tanto, el código del método sería el siguiente:

```
public Integer predecesor ( BTreeIF<Integer> bt ) {
     BTreeIF<Integer> node = bt.getLeftChild();
     if ( node == null ) { return null; }
     while ( node.getRightChild() != null ) {
          node = node.getRightChild();
     }
     return node.getRoot();
}
```

b) (0.5 Puntos) Calcule el coste asintótico temporal en el caso peor del método programado en el apartado anterior.

Se puede ver que todas las operaciones son de coste constante salvo el bucle, que dependerá de la longitud de la cadena de hijos derechos que tenga el hijo izquierdo de la raíz del árbol. Es decir, dependerá de su altura.

Por lo tanto, y sin conocer la relación entre el número de nodos del árbol y su altura, lo que se puede afirmar es que el coste estará en `O(h)`, siendo `h` la altura del árbol. Si bien es cierto que en el caso de un árbol degenerado, la altura `h` del árbol coincidirá con el número `n` de nodos del mismo, por lo que también sería posible afirmar que en el peor de los casos el coste estaría en `O(n)`.

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 ≤ 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 un interfaz `OrderedListIF<E>` de lista ordenada y, en caso positivo, cuáles serían los parámetros adecuados y si varían su precondición y postcondición con respecto a las de `ListIF<E>`.

En `ListIF<E>` nos encontramos con cuatro operaciones, así que vamos a analizarlas una por una:

- `get(int pos)`: esta operación devuelve el valor contenido en la posición `pos` de la lista. Tiene total cabida en un interfaz `OrderedListIF<E>` sin ningún cambio.
- `set(int pos, E e)`: esta operación modifica el valor contenido en la posición `pos` de la lista, cambiándolo por el valor `e`. Esto podría destruir la ordenación de la lista, por lo que esta operación no podría utilizarse en un interfaz `OrderedListIF<E>`, salvo que se especificase que tras la modificación la propia lista se encargaría de reordenar el elemento modificado, lo cual debería aparecer en la postcondición del método.
- `insert(E elem, int pos)`: esta operación inserta el valor `elem` en la posición `pos` de la lista. Tal y como está no podría aparecer en un interfaz `OrderedListIF<E>`, ya que nada garantiza que la posición en la que queremos insertar el elemento mantenga el orden. Para insertar elementos en una lista ordenada, deberíamos disponer de una operación que no reciba la posición donde se ha de insertar el elemento (`insert(E elem)`) en cuya postcondición se indique que el elemento será insertado de forma que la lista siga estando ordenada.
- `remove(int pos)`: esta operación elimina el valor contenido en la posición `pos` de la lista. Si la lista estaba originalmente ordenada, tras eliminar el elemento seguirá estando ordenada, por lo que esta operación es totalmente

válida para un interfaz `OrderedListIF<E>` sin ningún tipo de modificación.

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.

La única operación presente en `SequenceIF<E>` es la creación de un iterador, que secuencializa la estructura. Este proceso no se ve afectado por el hecho de que la lista esté ordenada o no, por lo que esta operación no se puede implementar más eficientemente.

En cuanto a las operaciones de `CollectionIF<E>`, sólo la operación `contains(E e)` podría implementarse más eficientemente si sabemos que la lista está ordenada, ya que cuando encontremos un valor que, según el orden, debería estar con posterioridad al valor buscado podemos cortar el proceso de búsqueda ante la certeza de que la lista no contiene dicho valor.

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?

En primer lugar, vamos a definir `n` como el tamaño de la lista y `s` como el tamaño máximo de las cadenas de caracteres.

Así, las operaciones que devuelven el tamaño de la lista y la vacían o comprueban si es vacía, todas ellas en `CollectionIF<E>`, seguirán teniendo un coste constante con respecto a ambos tamaños del problema, ya que pueden realizar su trabajo sin acceder a ningún elemento de la lista. También la operación que genera un iterador, ya que la lista es una secuencia.

La operación `contains` va a implicar un recorrido sobre los elementos de la lista y, para cada elemento, se tendrá que comparar la cadena buscada con la contenida en el elemento. Esta comparación (realizada internamente por Java) involucra una comparación uno a uno entre los caracteres de ambas secuencias, por lo que tendrá un coste lineal con respecto al tamaño de las cadenas de caracteres. Por lo tanto, teniendo en cuenta el recorrido sobre la lista, esta operación va a tener un coste que estará en $O(n \cdot s)$.

Las operaciones `get` y `remove` implican recorridos en la lista, pero no necesitan acceder al contenido de ningún elemento de la lista. En este caso, ambas operaciones tendrán un coste en $O(n)$.

La operación `insert`, al igual que `contains`, implica un recorrido sobre los elementos de la lista y para cada uno de ellos una comparación entre dos cadenas de caracteres para saber en qué posición se deberá insertar el elemento. Por el mismo argumento que el dado para `contains`, esta operación tendrá un coste en `O(n·s)`.

Dejamos para el final la operación `set(elem, pos)`, que recordemos sólo tendría sentido si tras la modificación del valor de un elemento, se encargase de recolocarlo en la lista. En este caso, se puede pensar que esa tarea es equivalente a eliminar el elemento de la posición `pos` y luego insertar un nuevo elemento `elem`. Por lo tanto, el coste sería la suma de ambos: `O(n) + O(n·s) = O(n·s)`.

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 ();
}
```

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);
    /* 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 <= pos <= 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 <= pos <= size()                                 */
    public void remove (int pos);
}
```

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);
}
```

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 ();
}
```