P1 (2 puntos) **Práctica**. Se desea añadir un método consultor al interfaz `QueryDepot` que devuelva, **en tiempo constante**, la consulta más frecuente (a igualdad de frecuencias, se podrá devolver cualquiera de las de mayor frecuencia) que esté almacenada en el depósito de consultas.

Indique cuáles serían los cambios en la estructura de datos y qué modificaciones habrían de hacerse en los métodos constructores y en los métodos de la interfaz `QueryDepot`, los cuales recordamos a continuación:

```
public interface QueryDepot {
  public int numQueries ();
  public int getFreqQuery (String q);
  public ListIF<Query> listOfQueries (String prefix);
  public void incFreqQuery (String q);
  public void decFreqQuery (String q);
}
```

1. Indique si son ciertas o falsas las siguientes afirmaciones sobre costes. En caso de que una afirmación sea cierta, deberá explicar el motivo, si fuera falsa deberá aportar un contraejemplo:

   a) (0'25 puntos) Un algoritmo de coste $\Theta(1)$ siempre tardará menos en ejecutarse que un algoritmo de coste $\Theta(2)$

   b) (0'25 puntos) Si el coste asintótico temporal de un algoritmo es $\mathcal{O}(\log n)$, entonces también es $\mathcal{O}(n)$

   c) (0'25 puntos) Un algoritmo de coste asintótico temporal $\mathcal{O}(n)$ puede tardar un tiempo arbitrariamente más grande en ejecutarse que otro del mismo coste asintótico temporal

   d) (0'25 puntos) Puede existir un tamaño de problema por debajo del cual es más eficiente usar el algoritmo de mayor coste asintótico temporal

2. La compresión RLE *(Run-length encoding)* es una forma de compresión de datos sin pérdidas en la que secuencias de datos consecutivos con el mismo valor se codifican mediante dicho valor y el número de veces que se repite. Por ejemplo, si aplicamos RLE a la siguiente secuencia de números:

   $$1-1-1-2-2-2-2-2-3-1-1-1$$

   obtendríamos la siguiente secuencia de pares: $(1,3)-(2,5)-(3,1)-(1,3)$, ya que la secuencia original comienza con tres repeticiones del 1, seguidas de cinco repeticiones del 2, seguidas de una repetición del 3 y finaliza con tres repeticiones del 1.

   Se desea crear un TAD que permita almacenar listas de elementos mediante compresión RLE, para ello se define la siguiente interfaz **`RLEListIF`**:

```
/* Representa una lista comprimida mediante RLE                    */
public interface RLEListIF<T> {

  /* [0'5 puntos] devuelve el número total de elementos de la lista */
  /* En el ejemplo debería devolver 12, no 4                        */
  public int size ();

  /* [1 punto] devuelve la lista descomprimida                      */
  public ListIF<T> decompress();

  /* [1 punto] calcula la moda (el elemento más repetido de la      *
   * lista). En el ejemplo, sería el 1, ya que se repite 6 veces    */
  public <T> mode ();
}
```

Consigne sus datos en todas las hojas que entregue. No se puntuarán respuestas sin justificar.

a) (1 punto) Describa detalladamente cómo realizaría la representación interna de este tipo (considere crear una clase interna para almacenar los pares de la lista comprimida). En base a esa representación interna, programe un constructor que, partiendo de un objeto `ListIF<T>` construya un elemento de una clase que implemente esta interfaz. Justifique su respuesta.

b) (2'5 puntos) Basándose en la respuesta a la pregunta anterior, implemente los métodos de la `interfaz RLEListIF<T>`. Se valorará la eficiencia. (Nota: las puntuaciones asignadas a cada método se indican en la especificación de dicho método en la interfaz del tipo).

c) (2'5 puntos) Programe una clase que implemente un iterador `IteratorIF<RLEListIF<T>>`, de forma que se permita el acceso a los elementos de la lista descomprimida sin utilizar el método `decompress()`. Detalle, si es necesaria, la representación interna del iterador, su constructor y los métodos `next()` y `hasNext()`.

d) (1 punto) Calcule el coste asintótico temporal, en el caso peor, del método `decompress()`. Justifique adecuadamente su respuesta.

ListIF (Lista)

```
/* Representa una lista de elementos */
public interface ListIF<T>{
   /* Devuelve la cabeza de una lista*/
    *
    public T getFirst ();
    /* Devuelve: la lista excluyendo la
       cabeza. No modifica la estructura */
    public ListIF<T> getTail ();
   /* Inserta una elemento (modifica la
      estructura)
    * Devuelve: la lista modificada
    * @param elem El elemento que hay que
       añadir*/
    public ListIF<T> insert (T elem);
    /* Devuelve: cierto si la lista esta
       vacia */
    public boolean isEmpty ();
    /* Devuelve: cierto si la lista esta
       llena*/
    public boolean isFull();
    /* Devuelve: el numero de elementos de
       la lista*/
    public int getLength ();
    /* Devuelve: cierto si la lista
       contiene el elemento.
     * @param elem El elemento buscado */
    public boolean contains (T elem);
    /* Ordena la lista (modifica la lista)
     * @Devuelve: la lista ordenada
     * @param comparator El comparador de
        elementos*/
    public ListIF<T> sort (ComparatorIF<T>
       comparator);
    /*Devuelve: un iterador para la lista*/
    public IteratorIF<T> getIterator ();
}
```

StackIF (Pila)

```
/* Representa una pila de elementos */
public interface StackIF <T>{
    /* Devuelve:  la cima de la pila */
     public T getTop ();
    /* Incluye un elemento en la cima de
       la pila (modifica la estructura)
     * Devuelve: la pila incluyendo el
        elemento
     * @param elem Elemento que se quiere
        añadir */
     public StackIF<T> push (T elem);
    /* Elimina la cima de la pila
       (modifica la estructura)
     * Devuelve: la pila excluyendo la
        cabeza */
     public StackIF<T> pop ();
    /* Devuelve: cierto si la pila esta
       vacia */
     public boolean isEmpty ();
    /* Devuelve: cierto si la pila esta
       llena */
     public boolean isFull();
    /* Devuelve: el numero de elementos de
       la pila */
     public int getLength ();
    /* Devuelve: cierto si la pila
       contiene el elemento
     * @param elem Elemento buscado */
     public boolean contains (T elem);
    /*Devuelve: un iterador para la pila*/
    public IteratorIF<T> getIterator ();
}
```

QueueIF (Cola)

```
/* Representa una cola de elementos */
public interface QueueIF <T>{
    /* Devuelve: la cabeza de la cola */
     public T getFirst ();
    /* Incluye un elemento al final de la
       cola (modifica la estructura)
     * Devuelve: la cola incluyendo el
        elemento
     * @param elem Elemento que se quiere
        añadir */
     public QueueIF<T> add (T elem);
    /* Elimina el principio de la cola
       (modifica la estructura)
     * Devuelve: la cola excluyendo la
        cabeza    */
     public QueueIF<T> remove ();
    /* Devuelve: cierto si la cola esta
       vacia */
     public boolean isEmpty ();
    /* Devuelve: cierto si la cola esta
       llena */
     public boolean isFull();
    /* Devuelve: el numero de elementos de
       la cola */
     public int getLength ();
    /* Devuelve: cierto si la cola
       contiene el elemento
     * @param elem elemento buscado */
     public boolean contains (T elem);
    /*Devuelve: un iterador para la cola*/
     public IteratorIF<T> getIterator ();
}
```

TreeIF (Árbol general)

```
/* Representa un arbol general de
   elementos */
public interface TreeIF <T>{
    public int PREORDER  = 0;
    public int INORDER   = 1;
    public int POSTORDER = 2;
    public int BREADTH   = 3;
    /* Devuelve:  elemento raiz del arbol
       */
     public T getRoot ();
    /* Devuelve: lista de  hijos de un
       arbol.*/
     public ListIF <TreeIF <T>>
        getChildren ();
    /* Establece el elemento raiz.
```

```
     * @param elem Elemento que se quiere
        poner como raiz*/
     public void setRoot (T element);
    /* Inserta un subarbol como ultimo hijo
     * @param child el hijo a insertar*/
     public void addChild (TreeIF<T>
        child);
    /* Elimina el subarbol hijo en la
       posicion index-esima
     * @param index indice del subarbol
        comenzando en 0*/
     public void removeChild (int index);
    /* Devuelve: cierto si el arbol es un
       nodo hoja*/
     public boolean isLeaf ();
    /* Devuelve: cierto si el arbol es
       vacio*/
     public boolean isEmpty ();
    /* Devuelve: cierto si la lista
       contiene el elemento
     * @param elem Elemento buscado*/
     public boolean contains (T element);
    /* Devuelve: un iterador para la lista
     * @param traversalType el tipo de
        recorrido, que
     * sera PREORDER, POSTORDER o BREADTH
        */
     public IteratorIF<T> getIterator (int
        traversalType);
}
```

BTreeIF (Árbol Binario)

```
/* Representa un arbol binario de
   elementos */
public interface BTreeIF <T>{
  public int PREORDER  = 0;
  public int INORDER   = 1;
  public int POSTORDER = 2;
  public int LRBREADTH = 3;
  public int RLBREADTH = 4;
 /* Devuelve: el elemento raiz del arbol */
  public T getRoot ();
 /* Devuelve: el subarbol izquierdo o null
    si no existe */
  public BTreeIF <T> getLeftChild ();
 /* Devuelve: el subarbol derecho o null
    si no existe */
  public BTreeIF <T> getRightChild ();
 /* Establece el elemento raiz
  * @param elem Elemento para poner en la
     raiz */
  public void setRoot (T elem);
 /* Establece el subarbol izquierdo
  * @param tree el arbol para poner como
     hijo izquierdo */
  public void setLeftChild (BTreeIF <T>
     tree);
 /* Establece el subarbol derecho
  * @param tree el arbol para poner como
     hijo derecho */
  public void setRightChild (BTreeIF <T>
     tree);
 /* Borra el subarbol izquierdo */
  public void removeLeftChild ();
 /* Borra el subarbol derecho */
  public void removeRightChild ();
 /* Devuelve: cierto si el arbol es un
    nodo hoja*/
  public boolean isLeaf ();
 /* Devuelve: cierto si el arbol es vacio
    */
  public boolean isEmpty ();
 /* Devuelve: cierto si el arbol contiene
    el elemento
  * @param elem Elemento buscado */
  public boolean contains (T elem);
 /* Devuelve un iterador para la lista.
  * @param traversalType el tipo de
     recorrido que sera
    PREORDER, POSTORDER, INORDER,
       LRBREADTH o RLBREADTH */
 public IteratorIF<T> getIterator (int
    traversalType);
}
```

ComparatorIF

```
/* Representa un comparador entre
   elementos */
public interface ComparatorIF<T>{
  public static int LESS  = -1;
  public static int EQUAL = 0;
  public static int GREATER = 1;
 /* Devuelve: el orden de los elementos
  * Compara dos elementos para indicar si
     el primero es
  * menor, igual o mayor que el segundo
     elemento
  * @param e1 el primer elemento
  * @param e2 el segundo elemento   */
  public int compare (T e1, T e2);
 /* Devuelve: cierto si un elemento es
    menor que otro
  * @param e1 el primer  elemento
  * @param e2 el segundo elemento */
  public boolean isLess (T e1, T e2);
 /* Devuelve: cierto si un elemento es
    igual que otro
  * @param e1 el primer  elemento
  * @param e2 el segundo elemento */
  public boolean isEqual (T e1, T e2);
 /* Devuelve: cierto si un elemento es
    mayor que otro
  * @param e1 el primer  elemento
  * @param e2 el segundo elemento*/
  public boolean isGreater (T e1, T e2);
}
```

IteratorIF

```
/* Representa un iterador sobre una
   abstraccion de datos */
public interface IteratorIF<T>{
    /* Devuelve: el siguiente elemento de
       la iteracion */
     public T getNext ();
```

```
    /* Devuelve: cierto si existen mas
       elementos en el iterador */
     public boolean hasNext ();
    /* Restablece el iterador para volver
       a recorrer la estructura */
     public void reset ();
}
```