Estrategias de Programación y Estructuras de Datos

Dpto. de Lenguajes y Sistemas Informáticos

Material permitido: NINGUNO. Duración: 2 horas

Alumno:	**D.N.I.:**
C. Asociado en que realizó la Práctica Obligatoria:	

Este documento detalla una posible solución propuesta por el Equipo Docente con intención didáctica antes que normativa. Nótese que el nivel de detalle de este documento no es el pretendido para los ejercicios realizados por los alumnos en la solución de sus exámenes.
Por brevedad, no se incluyen las interfaces de los tipos, que el alumno puede consultar en la documentación de la asignatura.

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);
}
```

La respuesta correcta a este apartado debe ser independiente de la representación utilizada (lista o árbol) y consisitiría en:

a) Modificar la estructura añadiendo un nuevo atributo de tipo `Query` que almacenase la consulta más frecuente.

b) Modificar los constructores para que inicializasen correctamente el atributo.

c) Modificar `incFreqQuery` para que comprobase si la frecuencia de la query que se le ha pasado como parámetro es superior a la de la query más frecuente, en cuyo caso debería actualizar el atributo.

d) Modificar `decFreqQuery` para que comprobase si la query a la que se le decrementa la frecuencia es la más frecuente, en cuyo caso habría que buscar en todo el depósito para volver a actualizar el atributo con la consulta más frecuente.

e) Los métodos consultores no necesitan modificación alguna.

Eso hace que el nuevo método consultor devuelva el valor pedido en un tiempo constante.

En cuanto a los **errores más frecuentes** en este apartado, han consistido en:

- Olvidar cualquiera de los puntos anteriormente enumerados, en especial, la necesidad de que el resultado se devolviese en tiempo constante (lo que exige un atributo que lo mantenga almacenado) y las consecuencias que esto genera para los métodos `incFreqQuery()` y `decFreqQuery()`, con especial atención al último por exigir una búsqueda de la nueva `Query` más frecuente.

- Realizar una solución particular para una sola de las implementaciones del tipo `QueryDepot`

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:

   La medida del coste que utilizamos en la asignatura (notación asintótica) es independiente de factores como el hardware utilizado para la ejecución del algoritmo, el sistema operativo en el que se ejecute, el lenguaje de programación en que se codifique o las unidades de medida.

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

   La afirmación es **falsa**.

   La notación de órdenes de complejidad oculta constantes, por lo que en realidad $\Theta(1)$ es el mismo conjunto que $\Theta(2)$: el conjunto de todas las funciones constantes.

   Lo único que podemos afirmar es que ambos algoritmos tienen un coste constante, pero no sabemos nada sobre las constantes multiplicativas de cada uno de ellos, por lo que la afirmación es falsa.

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

   La afirmación es **verdadera**.

   $\mathcal{O}(\log n)$ es el conjunto de todas las funciones con un crecimiento asintótico no superior al de $\log n$. De igual forma, $\mathcal{O}(n)$ es el conjunto de todas las funciones con un crecimiento asintótico no superior al de n.

   Dado que $\log n$ tiene un crecimiento asintótico no superior al de n, también $\log n$ está en el conjunto $\mathcal{O}(n)$, por lo que $\mathcal{O}(\log n)$ es un subconjunto de $\mathcal{O}(n)$.

   Eso significa que todo algoritmo con un coste asintótico en $\mathcal{O}(\log n)$, también es un algoritmo con un coste asintótico en $\mathcal{O}(n)$ y, por lo tanto, la afirmación es verdadera.

   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

   La afirmación es **verdadera**.

   Esta afirmación se refiere a las constantes multiplicativas. Dos algoritmos de coste lineal pueden diferir en una constante multiplicativa arbitrariamente grande, por lo que uno puede tardar un tiempo arbitrariamente más grande que el otro aún teniendo los dos el mismo coste asintótico temporal. Incluso, un algoritmo de coste constante *también* está en $\mathcal{O}(n)$, por lo que cualquier algoritmo de coste $\Theta(n)$ crecerá, asintóticamente hablando, infinitamente más que éste.

   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

   La afirmación es **verdadera**.

   En este caso habría que haber indicado que existe un umbral a partir del cual un algoritmo adquiere el comportamiento de su coste asintótico, pero antes puede tener un coste mayor.

   Otra alternativa podría haber sido poner un ejemplo en el que jugando con las constantes multiplicativas se viera cómo una función de coste asintóticamente peor que otra se comporta mejor por debajo de un determinado umbral.

En cuanto a los **errores más frecuentes** en este apartado, han consistido en respuestas cuyas *justificaciones* eran incorrectas, las más de las veces porque bien no comprendían la naturaleza asintótica del crecimiento (independencia de factores constantes, acercamiento progresivo a la asíntota a partir de un cierto valor para la entrada pero no necesariamente antes) bien porque no comprendían la naturaleza comparativa de la medida (independencia de factores tales como la máquina o el lenguaje de programación).

Nótese que no dar ninguna justificación o dar una *justificación* a una respuesta incorrecta son errores que anulan el subapartado en que se produzcan.

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 ();
}
```

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.

Como debemos relacionar cada elemento (un objeto de tipo genérico) con el número de veces que aparece repetido, parece sensato utilizar una lista de pares donde éstos se representarán, como sugería el enunciado como una clase interna por comodidad. Esta clase incluye los atributos necesarios y un constructor que toma siempre dos parámetros. Incluimos los preceptivos métodos consultores y modificadores de los atributos. En segunda opción y para facilitar la programación del iterador que se solicita en el apartado *c* siguiente, hemos decidido declarar esta clase (`RLEPair`) como pública.

`RLEPair`:

```
/* Representa un par para la compresión RLE. Se hace pública por
 * conveniencia para resolver la implementación de la clase iteradora */
public class RLEPair<T>{
  private T   element;
  private int freq;

  RLEPair (T e, int f){
    element = e;
    freq = f;
  }

  public T getElement(){
    return element;
  }
```

```
  public int getFreq(){
    return freq;
  }

  public void setElement(T e){
    element = e;
  }

  public void setFreq(int f){
    freq = f;
  }
}
```

En cuanto a la propia lista comprimida mediante RLE, su representación será, simplemente una lista ordenada (según aparición en la lista que se desea comprimir) de pares como los vistos anteriormente. Un estudio detallado de la interfaz propuesta para la lista RLE nos indica que ésta se va a construir una sola vez y nunca se va a modificar, ya que no hay métodos modificadores. Esto supone que los valores que deberán devolver las consultoras pueden calcularse en el constructor y consultarse en tiempo constante. De hecho, no es razonable, ya que induciría un coste desmesurado, calcular dichos valores mediante recorridos (véase la explicación en la enumeración de errores más comunes de este apartado al final del mismo).

Por lo tanto, incluiremos en la representación de la clase dos atributos que almacenen el tamaño y la moda respectivamente. Además, necesitaremos representar la lista de pares. En resumen la clase `RLEList`, que implementa la interfaz propuesta `RLEListIF` quedará como sigue (sólo la representación y el constructor, ya que los métodos de la interfaz se desarrollarán en el siguiente apartado):

`RLEListIF`:

```
/* Representa una lista comprimida mediante RLE (run-length          */
/* encoding)                                                         */
public class RLEList<T>{
  private int size;
  private T mode;
  private ListIF<RLEPair<T>> data;

  /* The empty constructor is included to make it easy setting up   */
  /* the initial values when actually constructing a compressed     */
  /* RLE list                                                       */
  public RLEList(){
    size = 0;
    mode = null;
    data = null;
  }

  /* Creates a Run-Length Encoded compressed list out of a list     */
  /* possibly including element repetitions and keeping the order   */
  /* of occurrence of the original elements                         */
  /* @param the list to be compressed                               */
  public RLEList(ListIF<T> input){
    this();
    size = input.getLength();
    data = compress (input, null, 0);
    mode = computeMode ();
  }
```

```
  /* Private method meant to carry out the actual compression       */
  /* @param the input (list) to be compressed                       */
  private RLEList<T> compress (ListIF<T> input, T elem, int count){
     // caso base: entrada vacía
     if (input.isEmpty()){
       ListDynamic<RLEPair<T>> init = new ListDynamic<RLEPair<T>>();
       if (elem == null)
         return init;
       RLEPair<T> aPair = new RLEPair(elem,count);
       return init.insert(aPair);
     }

     // Empieza el conteo de un nuevo grupo de elementos repetidos
     if (elem == null)
        return compress(input.getTail(),input.getFirst(),1);

     // Continúa el conteo de un grupo de elementos repetidos
     if (elem.equals(input.getFirst()))
       return compress (input.getTail(), elem, count++);

     // Termina el conteo de un grupo de elementos repetidos
     RLEPair<T> aPair = new RLEPair(elem,count);
     return compress (input.getTail(), null, 0).insert(aPair);
  }
}
```

En cuanto a los **errores más comunes** en este apartado, han consistido en:

- No detallar la representación del tipo o su(s) constructor(es) (lo que anula la calificación de la pregunta).
- Construir la lista RLE invirtiendo el orden de los valores en la lista parámetro original (nótese que la interfaz `ListIF` ofrece un método de inserción que coloca el nuevo elemento como primero de la lista cliente).
- No incluir atributos (y su cálculo **único** en el constructor) para el tamaño y la moda. Esta carencia (no incluir dichos atributos) obliga a realizar un recorrido de la lista para averiguar el tamaño y otro(s) para la moda, lo que supone un sobrecoste desmesurado: imaginemos que se comprime una lista de un millón de elementos-ahí ya se podrían calcular el tamaño y la moda, dado que ¡no van a cambiar nunca!-y luego se consulta mil veces cada uno de dichos valores. Si se precalculan, su coste será el de mil accesos. Si se hace como este error frecuentísimo describe (con un recorrido), mil millones.
- Utilizar **`int`** como tipo para los elementos de la lista en lugar de un tipo genérico (`T`).

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).

```
  /* Devuelve el número total de elementos de la lista original     */
  public int size (){
    return size;
  }

  /* Devuelve la moda (el elemento más repetido de la lista)        */
  public T mode (){
    return mode;
  }
```

```
/* Devuelve la lista descomprimida                                    */
public ListIF<T> decompress(){
  return expand (data, null, 0);
}
/* Private method to actually compute the mode                        */
private T computeMode (){
  return findMax (collapse ());
}
/* Private method to accumulate any repeated values on the same       */
/* RLE pair.                                                          */
private ListIF<RLEPair<T>> collapse (){
  ListIF<RLEPair<T>> auxList = new ListDynamic<RLEPair<T>> ();
  IteratorIF<ListIF<RLEPair<T>> dIter = data.getIterator ();
  RLEPair<T> auxPair, dPair;
  while (dIter.hasNext ()){
    IteratorIF<ListIF<RLEPair<T>> auxIter = auxList.getIterator ();
    dPair = iter.getNext ();
    boolean found = false;
    while (auxIter.hasNext () && !found){
      auxPair = auxIter.getNext ();
      found = (auxPair.getElement ().equals (dPair.getElement()));
    }
    if (found) auxPair.setFreq (auxPair.getFreq()+dPair.getFreq())
    else auxList.insert (dPair); /* comentar que no importa el orden */
  }
  return auxList;
}
/* Private method to find the most repeated element                   */
private T findMax (ListIF<RLEPair<T>> aList){
  RLEPair<T> aPair;
  RLEPair<T> mPair = new RLEPair (null, 0);
  IteratorIF<ListIF<RLEPair<T>> aIter = aList.getIterator ();
  while (aIter.hasNext()){
    aPair = aIter.getNext();
    if (aPair.getFreq()>mPair.getFreq()) mPair = aPair;
  }
  return mPair.getElement();
}
/* Private method to perform the actual decompression process         */
private ListIF<T> expand(ListIF<RLEPair<T>> rList, T e, int reps){
  if (reps == 0){
    if (rList.isEmpty())
       return rList;
    else {
      T elem   = rList.getFirst().getElement();
      int freq = rList.getFirst().getFreq();
      return expand (rList.getTail(), elem, freq);
    }
  }
  else {
    return expand (rList, e, reps-1).insert(e);
  }
}
```

Los **errores comunes** han consistido en:

- Descomprimir la lista invirtiendo el orden de los elementos de la la original (véase el anterior apartado).
- En el cálculo de la moda, olvidar el conteo de elementos iguales en pares separados de la lista.
- Almacenar la lista descomprimida para no necesitar realizar la descompresión. Obviamente, esto es un *fraude* al enunciado, ya que ¡se comprime para intentar reducir el espacio que ocupa la lista original!
- Devolver el tamaño de la lista comprimida, aunque el enunciado especificaba que se requería el de la lista original.

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()`.

Nótese que el constructor del iterador recibe la lista comprimida que se compone de pares `RLEPair<T>`. La necesidad de acceso a dichos pares desde la clase del constructor justifica que dicha clase se considere como pública en lugar de como interna/privada.

```
  /* Implements the RLEList iterator interface */
  public class RLEIterator<RLEPair<T>> implements IteratorIF<RLEPair<T>>{
    private RLEPair<T> iPair;
    private ListIF<RLEPair<T>> iList;

    public RLEIterator(ListIF<RLEPair<T>> rList){
      iPair = new RLEPair<T> (null, 0);
      iList = rList;
    }
    public hasNext(){
      return (iPair.getFreq()>0 || !iList.isEmpty());
    }
    /* En el material adicional, este método aparece como hasNext() */
    public next(){
      if (iPair.getFreq() == 0){
        if (iList.isEmpty()) return null;
        else {
          iPair = iList.getFirst();
          iList = iList.getTail();
        }
      }
      iPair.setFreq(iPair.getFreq()-1);
      return iPair.getElement();
    }
}
```

Los **errores comunes** han consistido en:

- Devolver un iterador para la estructura interna que almacena la lista.
- Almacenar la lista descomprimida (véase el anterior apartado) y devolver un iterador para dicha lista.
- Descomprimir la lista y devolver un iterador para dicho resultado, lo que estaba expresamente prohibido en el enunciado.

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

Cálculo del coste:

El tamaño del problema debe coincidir con el de la lista descomprimida, ya que ése es el que *crece* para diferentes ejemplares del problema.

El método `decompress()` realiza una llamada a `expand()`, que es el que lleva a cabo la descompresión real.

Se trata de un método recursivo y el coste de la parte no recursiva es constante, ya que sólo se realizan llamadas a métodos tales como `insert()` o `getTail()` de `ListIF`, que no dependen del tamaño de la lista cliente.

Nótese que para cada llamada externa se produce una única llamada recursiva interna. El tamaño del problema decrece en una unidad (por substracción) para cada nueva invocación y, como ya justificamos anteriormente, el coste de la parte no recursiva es constante. En estas condiciones, el coste del método `expand()` es lineal respecto al número de elementos de la lista original.

Como nota curiosa, véase que, en realidad, se producen más invocaciones que elementos tiene dicha lista original. Esto sucede porque hay que hacer una doble recursión respecto a los pares distintos y al número de repeticiones en cada uno de ellos. Como máximo, el número de invocaciones internas sería el doble del tamaño de la lista original, en el caso extremo de que cada par tuviese exactamente un elemento (frecuencia 1).

Los **errores comunes** han sido:

- Como siempre, ya que se trata del error más común de la historia de esta asignatura (y de otras pasadas), el principal se trata de **no detallar el tamaño del problema**. Un coste **siempre** requiere una unidad de medida. Imagínese tratar de adquirir un bien de consumo o contratar un servicio y preguntar el precio. ¿En qué *moneda*?¿Euros, dólares americanos, libras esterlinas o yenes japoneses?. Pues esto es lo mismo. La especificación del tamaño es *imprescindible*. No incluirla anula la calificación del apartado de manera inmediata. Este tamaño debe explicarse o justificarse y debe ser un parámetro que dependa del problema, no de la solución y que sea variable en función de la entrada de datos del problema.
- En segundo lugar, no calcular o, cuando menos, justificar el coste. No basta con decir *es lineal*. Hay que explicar por qué lo es o en función de qué factores.
- Algo menos frecuente pero aún patente ha sido *obtener* un coste que no tiene relación con el método implementado. El coste en sí no es positivo o negativo. Existen algoritmos cuyo coste es *naturalmente* alto (por ejemplo, un polinómico de grado elevado o, incluso, un exponencial). Esto nos puede parecer peor solución pero la falta de coherencia entre *la solución* y su *coste* es inadmisible.