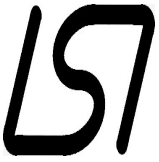


Estrategias de Programación y Estructuras de Datos

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 Práctica. Se quiere mantener un histórico que contenga exclusivamente las últimas n queries añadidas al depósito, donde n es un número conocido pasado por parámetro. Se pide:

- (a) (0.5 puntos) Describir cómo puede representarse dicho histórico mediante los TADs estudiados en la asignatura justificando razonablemente su respuesta.

  Se pide representar mediante los TADs de la asignatura el histórico que almacena las últimas n queries añadidas al depósito justificando la respuesta. El histórico debería representarse mediante una **cola**, que debería ser **estática de longitud n**, por conocerse previamente el número de queries que quieren almacenarse.

  La justificación pasa porque usando una cola estática de dicha longitud, no sólo podemos almacenar las n últimas queries introducidas en el depósito, sino que también controlamos el orden en el que entraron, de manera que saldrá primero de la cola la primera en entrar. Esto es especialmente útil para el caso en el que el histórico esté lleno y se requiera eliminar la primera query almacenada en el histórico e introducir una nueva que acaba de almacenarse en el depósito.

  En cuanto a los **errores comunes**, el principal ha consistido en una mala elección de la estructura de datos: El histórico sigue una política tipo FIFO (*first in, first out*) como se ha explicado anteriormente. El TAD cola es la estructura de datos que implementa dicha política.

- (b) (1.5 puntos) Identificar las operaciones de `QueryDepot` que se ven afectadas por este cambio y detallar cómo deberán modificarse (no es necesario que las implemente).

  La introducción del depósito hace variar:

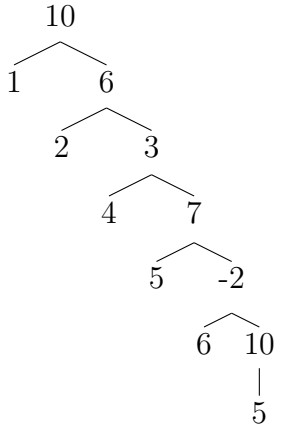
  - El constructor: se le debe pasar un entero que especifique el número n de queries que almacenará el histórico.
  - `incrFrecQuery`: Es la operación usada para introducir nuevas queries en el depósito, o incrementar la frecuencia de aquellas ya existente en el mismo. Su modificación pasa por introducir la query pasada por parámetro en el histórico. En caso de que el histórico esté lleno, previamente se ha de desencolar, de manera que se elimina la query que primero entró en el depósito perteneciente también al histórico.
  - El resto de operaciones del TAD no se ven afectadas por la incorporación del histórico.

  Los **errores comunes** han sido:

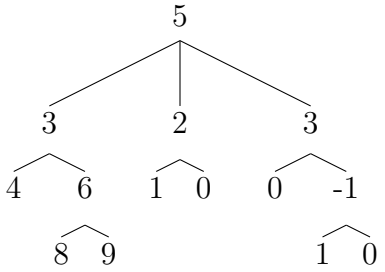
  - No señalar las operaciones dichas con anterioridad.
  - No explicar con detalle cómo modificar el método `incrFrecQuery`.

1. (2 puntos) Diremos que un árbol general no vacío está *completamente desnivelado* si no contiene dos hojas situadas en el mismo nivel. Por ejemplo, los árboles (a) y (c) son *completamente desnivelados* pero el (b) no lo es.

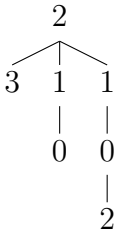
(a)



(b)



(c)



Se pide implementar en JAVA dentro del tipo `TreeIF` un método `bool isFullyUneven()` que decida si un árbol general está completamente desnivelado. Si lo considera preciso, utilice estructuras de datos adicionales que se correspondan con las vistas en la asignatura. Implemente métodos auxiliares si lo considera necesario para que su código sea lo más modular y claro posible.

Se pide diseñar un método que decida si un árbol general no vacío está completamente desnivelado o no. Se entiende que un árbol general no vacío está completamente desnivelado si en ninguno de sus niveles hay más de una hoja.

Una posibilidad consiste en analizar los arboles de cada nivel (que son hijos de los del nivel anterior) y contar el número de hojas. En caso de que en alguno de los niveles haya más de 1, devolver false, y en caso de que no se de esa condición en ninguno de los niveles del árbol, devolver `true`. Nótese que un nivel de un árbol, según el TAD `TreeIF`, puede verse como una lista de árboles.

Para presentar un código más modular, introducimos un par de métodos auxiliares que declararemos como privados:

```
// Devuelve el número de hojas del nivel de un árbol (lista de árboles)
private int numLeaves(ListIF<TreeIF<T>> level){
  int leaves = 0;
  Iterator<TreeIF<T>> iter=level.getIterator();
  while(iter.hasNext()){
    TreeIF<T> treeAux = iter.getNext();
    if (tree Aux.isLeaf()) leaves++;
  }
  return leaves;}
// Dado el nivel de un árbol, devuelve su siguiente nivel.
private ListIF<TreeIF<T>> getNextLevel(ListIF<TreeIF<T>> level){
  ListIF<TreeIF<T>> nextLevel = new  ListIF<TreeIF<T>> ();
  Iterator<TreeIF<T>> iter=level.getIterator();
  while(iter.hasNext()){
    TreeIF<T> treeAux = iter.getNext();
    ListIF<TreeIF<T>> children = treeAux.getChildren();
    Iterator<TreeIF<T>> iter2=children.getIterator();
    while(iter2.hasNext()){
      TreeIF<T> son = iter2.getNext();
      nextLevel.insert(son);
    }
  }
  return nextLevel;
}
```

Con los anteriores métodos auxiliaries podemos crear el método pedido en el enunciado

```
public boolean isFullyUneven(){
  if(isEmpty()){
    return false;
  }
  ListIF<TreeIF<T>> currentLevel = getChildren();
  while (!currentLevel.isEmpty()){
    int leaves = numLeaves(currentLevel);
    if(leaves>1){
      return false;
    }
    currentLevel = getNextLevel(currentLevel);
  }
  return true;
}
```

Nótese qué en el caso de tener un árbol consistente en un nodo hoja, el método devuelve true porque trivialmente cumple que no hay más de una hoja en el mismo nivel. El Equipo Docente ha tenido en cuenta otras posibles soluciones a la hora de corregir este ejercicio. En cuanto al coste del algoritmo anterior: el bucle da tantas vueltas como número h de niveles tiene el árbol. El método más costoso dentro del bucle es `getNextLevel()`, que realiza recorrido tanto del número de nodos del nivel actual como del nivel siguiente. Suponiendo que ambos valores son idénticos, sea k, entonces el anterior algoritmo tiene un coste de $\mathcal{O}(hk^2)$

Los **errores comunes** han consistido en

- No entender la definición de árbol general completamente desnivelado. En varios casos, los estudiantes miran el número de hijos, pero no el de hojas, y en base a ello deciden si el árbol es o no completamente desnivelado.
- Incorrecto empleo (léase recorrido, etc.) del árbol (e.g. mal uso de iteradores).
- Incorrecto empleo de recursión (e.g. un método se llama a sí mismo de manera que jamas termina)
- Olvidar casos base.

2. En un periódico digital, los artículos se organizan en secciones y además se etiquetan con palabras clave. Cada artículo se compone del texto con su contenido, una única sección a la que pertenece, y una o más palabras clave. Nos interesa definir la búsqueda por secciones y palabras clave, en la que el usuario puede introducir cualquier combinación de estas: sólo una sección, sólo una o más palabras clave, o ambas. Para dar más flexibilidad a la consulta, la sección puede aparecer en cualquier lugar de la consulta (antes, después, o intercalada en la lista de palabras clave). Podemos definir la siguiente interfaz:

```
// Representa un artículo
public interface ArticleIF {

  /* Devuelve el contenido del artículo                */
  public String getContent ();
  /* Devuelve las etiquetas asociadas al artículo      */
  public ListIF <String> getTags ();
  /* Devuelve la sección a la que pertenece el articulo */
  public String getSection ();
}
```

```
// Representa el periódico digital
public interface NewsPaperIF {

/** Añadir una noticia al periódico
* @param content: texto del artículo
* @param section: sección del artículo
* @param keywords: etiquetas del artículo
*/
public void addArticle (String content, String section, ListIF<String>
   keywords)

/** búsqueda  de artículos con unas etiquetas  determinadas y/o       */
  * incluidos en una sección determinada                               */
  * @param tags: lista de etiquetas de búsqueda (una puede ser la     */
  * sección)                                                          */
public ListIF <ArticleIF> getArticles (ListIF <String> tags);
}
```

Supondremos que existe una clase `Article` que implementa `ArticleIF` completamente. Consideremos dos implementaciones alternativas para `NewsPaperIF`: en la primera (`NewsPaperList`), el periódico se representará simplemente como una lista de objetos de tipo `Article`; en la segunda (`NewsPaperIndex`), se asociará un identificador (un entero) para cada artículo, y para cada etiqueta se creará una lista con los identificadores de los artículos en los que aparece.

Se pide:

a) (0.5 puntos) Detalle el constructor por defecto y el constructor por parámetros (recibiendo una lista inicial de artículos) de `NewsPaperList`.

La representación de `NewsPaperList` se describía explícitamente en el enunciado. Bastaba con tomar una lista de objetos tipo `Article`.

```
public class NewsPaperList{
  public ListDynamic<Article> newspaper;
  ...
  /* Los constructores pueden implementarse de la siguiente     */
  /* manera:                                                     */
  /* Constructor por defecto de NewsPaperList (0.25 puntos)     */
  NewsPaperList(){
    /* Construye un periódico vacío representado por una lista  */
    /* vacía de artículos                                       */
    newspaper = new ListDynamic<Article>();
  }
  // Constructor por parámetros de NewsPaperList (0.25 puntos)
  NewsPaperList(ListIF<Article> L){
  /* Construye el periódico a partir de la lista de artículos   */
  /* dada por parámetro                                         */
  newspaper = new ListDynamic<Article>();
  Iterator<Article> iter = L.getIterator();
  while(iter.hasNext()){
    Article A = iter.getNext();
    // vale newspaper.insert(A);
    addArticle(A.getContent(), A.getSection(), A.getTags());
  }
}
```

Aunque, por lo general, los estudiantes han respondido correctamente a este apartado, los **errores comunes** han consistido en que algunos no han representado el periódico siguiendo lo especificado en el enunciado.

b) (1 punto) Detalle el constructor por defecto y el constructor por parámetros (recibiendo una lista inicial de artículos) de `NewsPaperIndex`.

La representación de `NewsPaperIndex` se describía explícitamente en el enunciado: utilización de una estructura que ligase cada etiqueta con el conjunto de artículos que la contienen. Para ello se requiere de una clase auxiliar que asocie a cada etiqueta una lista de artículos que la contienen:

```
public class Index{
  public String tag;
  public ListDynamic<Article> articles;
  ...
}
```

Los constructores y métodos accesores y modificadores (getters y setters) de esta clase auxiliar se pueden dar por supuestos. Aunque no se implemente, hay que decir que se requiere un método **public void** `addArticle(Article A)` tal que añada el artículo pasado por parámetro a la lista de artículos, y cuya implementación simplemente consiste en insertar el artículo dado por parámetro a la lista de artículos, i.e., `articles.insert(A);`. La clase `NewsPaperIndex` puede representarse mediante una lista de objetos tipo `Index`, de manera que cada objeto `Index` de la lista se corresponda con etiquetas diferentes.

```
public class NewsPaperIndex{
  public ListDynamic<Index> newspaper;
  ...
}
```

Los constructores pueden implementarse de la misma manera que en NewsPaperList salvo por la línea `newspaper = new ListDynamic<Article>();`, que pasa a ser `newspaper = new ListDynamic<Index>();`.Sin embargo, el método `addArticle` en esta implementación, debe actualizar convenientemente los elementos de tipo `Index` contenidos por el periódico.

Los **errores comunes** han consistido en no representar el periódico mediante una estructura que asocie etiquetas con los artículos que las contienen tal y como se pedía, explícitamente, en el enunciado del examen. Se recomienda siempre pensar previamente qué información se ha de guardar y en qué tipo de estructura de datos. El empleo de clases auxiliares no supone ningún problema siempre y cuando se usen los TADs de la asignatura. Además, se dan por supuestos métodos triviales como los comentados en la solución de este apartado.

c) (1.5 puntos) Implemente en JAVA las operaciones `addArticle()` y `getArticles()` para la clase `NewsPaperList`.

```
// Implementación de las operaciones en NewsPaperList
// addArticle()  (0.5 puntos)
public void addArticle(String content, String section, List<String>
   tags) {
  Article A = new Article(content,section,tags);
  newspaper.insert(A);
}
```

```
//getArticles()  (1 punto)
public ListIF<Article> getArticles(List<String> tags){
  ListIF<Article> articles = new ListIF<Article>();
  Iterator<Article> iter = newspaper.getIterator();
  while(iter.hasNext()){
    Article A = iter.getNext();
    //comprobación por sección de artículo
    if (tags.contains(A.getSection())){
      articles.insert(A);
    }
    else{
      //comprobación por etiquetas de artículo
      ListIF<String> aTags = A.getTags();
      Iterator<String> iter2 = aTags.getIterator();
      Boolean centinela = true;
      while (centinela && iter2.hasNext()){
        String tagAux = iter2.getNext();
        If(tags.contains(tagAux)){
          articles.insert(A);
          centinela = false;
        }
      }
    }
  }
  return articles;
}
```

Los **errores comunes** han consistido en

- En general, los estudiantes han respondido correctamente la parte del método `addArticle()`.
- En cuanto el método `getArticles()`:
  - Olvidar comparar por sección.
  - Recorrer incorrectamente las estructuras de datos.

d) (2 puntos) Implemente en JAVA las operaciones `addArticle()` y `getArticles()` para la clase `NewsPaperIndex()`.

El siguiente método auxiliar devuelve el objeto tipo `Index` asociado a la etiqueta que recibe por parámetro. En caso de que no exista, devuelve **`null`**. Este mecanismo permite usar este método para decidir si existe tal objeto de tipo `Index` en el periódico.

```
public Index getIndex(String tag){
  Iterator<Index> iter = newspaper.getIterator();
  while(iter.hasNext){
    Index indexAux=iter.getNext();
    String tagIndex=indexAux.getTag();
    if(tag.equals(tagIndex)){
      return indexAux;
    }
  }
  return null;
}
```

```
// Implementación de las operaciones en NewsPaperIndex
//addArticle (1 punto)
public void addArticle(String content, String section, List<String>
   tags) {
  Article A = new Article(content,section,tags);
  Iterator<String> iter = tags.getIterator();
  while(iter.hasNext()){
    String tagAux = iter.getNext();
    Index index = getIndex(tagAux);
    If(index!=null){
      //agregar artículo al índice de la etiqueta
      index.addArticle(A);
    }
    else{
      //nueva etiqueta -> crear nuevo índice con el artículo
      ListIF<Article> L=new ListIF<Article>();
      L.insert(A);
      Index newIndex= new Index(tagAux,L);
      newspaper.insert(newIndex);
    }
  }
}

//getArticles (1 punto)
public ListIF<Article> getArticles(ListIF<String> tags){
  /* leer las etiquetas dadas por parámetro, tomar sus artículos */
  /* asociados, e insertarlos en la lista devuelta por el método */
  ListIF<Article> articles = new ListIF<Article>();
  Iterator<String> iter = tags.getIterator();
  while(iter.hadNext()){
    String tagAux = iter.getNext();
    Index index=getIndex(tagAux);
    If(index!=null){
      ListIF<Article> articlesIndex = index.getArticles();
      Iterator<Article> iter2 = articlesIndex.getIterator();
      while(iter2.hasNext()){
        Article A = iter2.getNext();
        articles.insert(A);
      }
    }
  }
  return articles;
}
```

Los **errores comunes** han consistido en

- En los casos donde los estudiantes no representaron adecuadamente el periódico según lo especificado en el enunciado, los errores en este apartado se derivan de la incorrecta representación.
- En los casos en los que los estudiantes han representado correctamente el periódico, los errores existentes se han debido al actualizar incorrectamente la estructura de datos al añadir artículos, y errores debidos a un mal uso/recorrido de las estructuras de datos.

e) (1 punto) Compare el coste (en tiempo y memoria) de las dos implementaciones. ¿En qué condiciones sería más eficiente optar por `NewsPaperIndex`, y en cuáles por `NewsPaperList`?

Dedicaremos un punto al coste temporal y otro al coste espacial (coste en memoria) para, en un tercer punto, comparar ambas soluciones:

1) Coste temporal:
   - `NewsPaperList`:
     - `addArticle()`: $\mathcal{O}(1)$ - coste de insert.
     - `getArticles()`: Sea n el número de artículos del periódico, y k el número de etiquetas del parámetro. Se recorren los n artículos, y en el caso peor, para cada ellos, se mira si contienen alguna de las etiquetas, de forma que se han de recorrer las k etiquetas. Además, para comprobar si efectivamente cada etiqueta se contiene en el artículo, se usa el método `contains()` dentro del segundo bucle, cuyo coste es líneal (sobre el número de etiquetas). Suponiendo que los artículos contienen k etiquetas, el coste está en $\mathcal{O}(nk^2)$.
   - `NewsPaperIndex`:
     - `addArticles()`: Sea k el número de etiquetas en la lista dada por parámetro. El método recorre dichas etiquetas, luego su coste está en $\mathcal{O}(k)$.
     - `getArticles()`: Sea k el número de etiquetas en la lista dada por parámetro, m el número de etiquetas contenidas en el periódico (objetos tipo `Index`). Se recorren las k etiquetas de la lista, y para cada una de de ellas se mira si existe una entrada `Index` (método `getIndex()` de coste $\mathcal{O}(m)$). En caso afirmativo se recorren los artículos contenidos dicho índice, que el caso peor serían todos los n artículos. Por tanto, hay 3 bucles anidados y el coste del método está en $\mathcal{O}(k \times m \times n)$.
2) Coste en memoria: sea n el número de artículos y m el número total de etiquetas:
   - `NewsPaperList`: No depende de las etiquetas, luego: $\mathcal{O}(n)$
   - `NewsPaperIndex`: Contiene tantas listas como etiquetas (m) y por otro lado almacena los artículos (n). Las listas contienen referencias a los artículos, de manera que el coste está en $\mathcal{O}(n+m)$, es decir, $\mathcal{O}(max(n,m))$
3) Comparativa entre las dos implementaciones.
   Las búsquedas de artículos en `NewsPaperList` dependen del número total de artículos, n contenidos en el periódico. En cambio, para `NewsPaperIndex`, esas búsquedas dependen del número k de etiquetas que proporciona el usuario para buscar artículos de su interés. Puede suponerse que este último número es menor que el número m total de etiquetas, i.e. $k < m$, y también es razonable pensar que un periódico, hay más artículos que etiquetas pone un usuario a la hora de buscar artículos de su interés. Por lo dicho anteriormente, la implementación con `NewsPaperIndex` es más adecuada para realizar búsquedas de artículos para periódicos que contienen muchos artículos. En cambio, la implementación con `NewsPaperList` sería más adecuada para periódicos que contienen pocos artículos. Suponiendo el caso de periódicos con pocos artículos, el sistema de búsqueda tendría menos sentido porque los usuarios tendrían menos contenido por el que navegar a través del periódico. Por tanto, es más útil la implementación planteada en `NewsPaperIndex`.

Los **errores comunes** han consistido en

- No indicar en función de qué tamaño se calculan los costes.
- Mal cálculo de la complejidad computacional de los métodos: no se corresponden con las implementaciones realizadas.