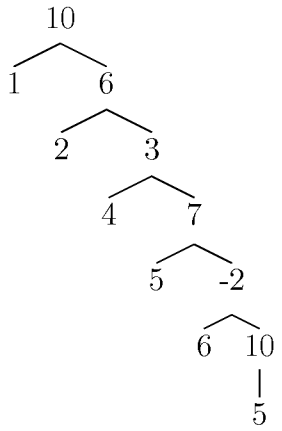
P1 Práctica. Se quiere mantener un histórico que contenga exclusivamente las últimas n queries añadidas al depósito, donde n es un número conocido pasado por parámetro. Se pide:

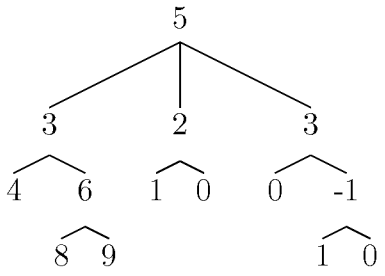
- (a) (0.5 puntos) Describir cómo puede representarse dicho histórico mediante los TADs estudiados en la asignatura justificando razonablemente su respuesta.
- (b) (1.5 puntos) Identificar las operaciones de `QueryDepot` que se ven afectadas por este cambio y detallar cómo deberán modificarse (no es necesario que las implemente).

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.

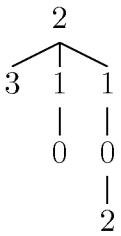
(a)



(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.

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`.

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

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

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

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`?

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 ();
}
```