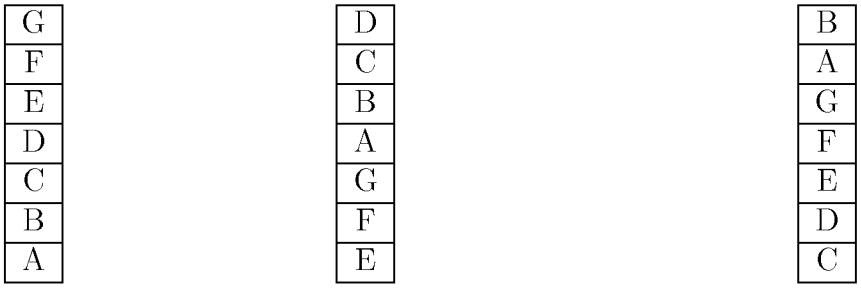
P1 Práctica. En el escenario simplificado se desea añadir una nueva operación `distance(int id)` a la clase `DoctorS`, que calcule la distancia que le separa del doctor cuyo identificador recibe como parámetro. Dicha distancia se calcula sumando las generaciones que separan a cada uno de los doctores de su primer ancestro común (entendiendo que un doctor es ancestro de sí mismo en 0 generaciones).

a) (1 punto) Describa detalladamente (sin necesidad de implementar) cómo realizaría la implementación del método `distance(int id)` en la clase `DoctorS`.

b) (1 punto) El cálculo de la distancia descrito no es válido para el escenario completo. De un ejemplo en el que se demuestre esta afirmación y describa cómo se debería (y bajo qué condiciones se podría) realizar el cálculo de la distancia en este escenario.

1. Se desea enriquecer el TAD pila añadiendo dos métodos que roten su contenido hacia arriba y hacia abajo un número determinado de veces. Por ejemplo:



Pila original Pila rotada hacia arriba 3 veces Pila rotada hacia abajo 2 veces

Si el número de rotaciones fuese 0, no se produciría ninguna rotación y si fuese negativo, se debería rotar en sentido contrario. Por ejemplo, rotar -3 veces hacia arriba es equivalente a rotar 3 veces hacia abajo. Por lo tanto, es perfectamente válido que, en caso necesario, el método de rotación hacia arriba pueda llamar al método de rotación hacia abajo y viceversa.

a) (0'75 puntos) Programar un método `ru(int)` en el TAD `StackIF<T>` que sea capaz de rotar el contenido de la pila hacia arriba un número determinado de veces cumpliendo lo indicado en el enunciado.

b) (0'75 puntos) Programar un método `rd(int)` en el TAD `StackIF<T>` que sea capaz de rotar el contenido de la pila hacia abajo un número determinado de veces cumpliendo lo indicado en el enunciado.

c) (0'5 puntos) Calcule el coste asintótico temporal en el caso peor del método `rd(int)`. Ponga especial atención en la definición del tamaño del problema.

Se valorará la eficiencia de los métodos implementados.

2. Un Álbum de cromos puede ser visto como un conjunto de cromos indexados entre 1 y n (siendo n el número de cromos total del ábum). Durante el proceso de coleccionar los cromos de un álbum se desea almacenar la información sobre el número de ejemplares que se posee de cada cromo individual, de forma que se pueda automatizar el proceso de intercambio de cromos repetidos.

Clase **Album**:

```
/* Representa un álbum de cromos                              */
public class Album{
  /* Devuelve el número de cromos del álbum [0'5 puntos]   *
   * @return un entero representando el número de cromos. */
  public Integer size();
```

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

```
  /* Añade cromos a la colección del álbum   [0'5 puntos] *
   * @param Lista de cromos (representados por sus        *
   *        índices dentro del álbum) para añadir.        */
  public void addCards(ListIF<Integer>);
  /* Comprueba si el álbum se ha completado [0'75 puntos] *
   * @return un valor booleano que indica si ya se posee  *
   *         al menos un ejemplar de cada cromo del álbum */
  public bool isComplete();
  /* Devuelve la lista de los cromos que faltan para      *
   * completar el álbum.                  [0'75 puntos]   *
   * @return lista de cromos (representados por sus       *
   *         índices dentro del álbum) de los que no se   *
   *         tiene ningún ejemplar                        */
  public ListIF<Integer> missingCards();
  /* Devuelve un objeto Trade con un posible intercambio  *
   * de cromos                                 [1 punto]  */
   * @param un álbum de cromos con el que intercambiar    *
   * @return dos listas: una con los cromos repetidos del *
   *         álbum llamante que no están en el álbum      *
   *         parámetro y otra con los cromos repetidos    *
   *         álbum parámetro que no están en el álbum     *
   *         llamante.                                    *
   * @pre this.size() == a.size                          */
  public Trade tradingCards (Album a);
}

Trade:
/* Representa un posible intercambio de cromos           */
 public class Trade{
   private ListIF<Integer> myRepeatedCards;
   private ListIF<Integer> myMissingCards;
/* Añade un cromo repetido al intercambio [0,5 puntos]   *
 * @param Índice del cromo que se desea añadir           */
 public addRepeatedCard(Integer id);
 /* Añade un cromo que falta al intercambio [0,5 puntos] *
 * @param Índice del cromo que se desea añadir           */
 public addMissingCard(Integer id);
}
```

Para la clase `Trade` asumiremos la existencia de un constructor que no recibe ningún parámetro. Se pide:

a) (1 punto) Describa detalladamente cómo realizaría la representación interna del tipo de datos `Album` usando los TAD vistos en la asignatura. Justifique su respuesta. Implemente un constructor de clase que reciba un entero con el número de cromos del Álbum y otro que reciba, además, una lista de índices de cromos declarada `ListIF<Integer>`.

b) (3'5 puntos) Implemente en Java los métodos descritos en la clase `Album`. Las puntuaciones están detalladas en la descripción de los métodos. Se valorará la eficiencia.

c) (1 punto) Implemente en Java los métodos descritos en la clase `Trade`. Las puntuaciones están detalladas en la descripción de los métodos. Se valorará la eficiencia.

d) (0,5 puntos) Calcule, de forma razonada, el coste asintótico temporal en el caso peor del método `tradingCards(Album)`.

CollectionIF (Colección)

```
/* Representa una colección de elementos. Una colección no          *
 * tiene orden.                                                     */
public interface CollectionIF<E> {
    /* Devuelve el número de elementos de la colección.             *
     * @return: cardinalidad de la colección.                       */
    public int size ();
    /* Determina si la colección está vacía.                        *
     * @return: size () == 0                                        */
    public boolean isEmpty ();
    /* Determina la pertenencia del parámetro a la colección        *
     * @param: el elemento cuya pertenencia se comprueba.           *
     * @return:param \in self                                       */
    public boolean contains (E e);
    /* Elimina todos los elementos de la colección.                 */
    public void clear ();
    /* Devuelve un iterador sobre la colección.                     *
     * @return: un objeto iterador para los elementos de            *
     * la colección.                                                */
    public IteratorIF<E> iterator ();
}
```

SetIF (Conjunto)

```
/* Representa un conjunto de elementos. Se trata del concepto       *
 * matemático de conjunto finito (no tiene orden).                  */
public interface SetIF<E> extends CollectionIF<E> {
    /* Devuelve la unión del conjunto llamante con el parámetro     *
     * @param: el conjunto con el que realizar la unión             *
     * @return: self \cup @param                                    */
    public SetIF<E> union (SetIF<E> s);
    /* Devuelve la intersección con el parámetro.                   *
     * @param: el conjunto con el que realizar la intersección.     *
     * @return: self \cap @param                                    */
    public SetIF<E> intersection (SetIF<E> s);
    /* Devuelve la diferencia con el parámetro (los elementos       *
     * que están en el llamante pero no en el parámetro).           *
     * @param: el conjunto con el que realizar la diferencia.       *
     * @return: self \setminus @param                               */
    public SetIF<E> difference (SetIF<E> s);
    /* Determina si el parámetro es un subconjunto del llamante.    *
     * @param: el posible subconjunto del llamante.                 *
     * @return: self \subseteq @param                               */
    public boolean isSubset (SetIF<E> s);
}
```

ListIF (Lista)

```
/* Representa una lista de elementos.                               */
public interface ListIF<E> extends CollectionIF<E>{
    /* Devuelve el elemento de la lista que ocupa la posición       *
     * indicada por el parámetro.                                   *
     * @param pos la posición comenzando en 1.                      *
     * @Pre: 1 \leq pos \leq 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 \leq pos \leq 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 \leq pos \leq 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 \leq pos \leq size()                                  */
    public void remove (int pos);
}
```

StackIF (Pila)

```
/* Representa una pila de elementos.                                */
public interface StackIF <E> extends CollectionIF<E>{
    /* Obtiene el elemento en la cima de la pila                     *
     * @Pre !isEmpty ();                                             *
     * @return la cima de la pila.                                   */
     public E getTop ();
    /* Incluye un elemento en la cima de la pila. Modifica el        *
     * tamaño de la misma.                                          *
     * @param elem el elemento que se quiere añadir en la cima       */
     public void push (E elem);
    /* Elimina la cima de la pila. Modifica el tamaño de la          *
     * pila.                                                        *
     * @Pre !isEmpty ();                                             */
     public void pop ();
}
```

QueueIF (Cola)

```
/* Representa una cola de elementos.                                */
public interface QueueIF <E> extends CollectionIF<E>{
    /* Devuelve el primer elemento de la cola.                       *
     * @Pre !isEmpty()                                               *
     * @return la cabeza de la cola (su primer elemento).            */
    public E getFirst ();
    /* Incluye un elemento al final de la cola. Modifica el          *
     * tamaño de la misma (crece en una unidad).                    *
     * @param elem el elemento que debe encolar (añadir).            */
    public void enqueue (E elem);
    /* Elimina el primer elemento de la cola. Modifica la            *
     * tamaño de la misma (decrece en una unidad).                  *
     * @Pre !isEmpty();                                              */
    public void dequeue ();
}
```

TreeIF (Árbol general)

```
/* Representa un árbol n-ario de elementos, donde el número de    *
```

```
 * hijos de un determinado nodo no está determinado de antemano  *
 * (fan-out no prefijado, no necesariamente igual en cada nodo). */
public interface TreeIF<E> extends CollectionIF<E>{
    public int PREORDER  = 0;
    public int POSTORDER = 1;
    public int BREADTH   = 2;
    /* Obtiene la raíz del árbol (único elemento sin antecesor). *
     * @Pre: !isEmpty ();                                        *
     * @return el elemento que ocupa la raíz del árbol.          */
     public E getRoot ();
    /* Modifica la raíz del árbol.                               *
     * @param el elemento que se quiere poner como raíz del      *
     * árbol.                                                    */
     public void setRoot (E e);
    /* Obtiene los hijos del árbol llamante.                     *
     * @Pre: !isEmpty ();                                        *
     * @return la lista de hijos del árbol (en el orden en que   *
     * están almacenados en el mismo).                           */
     public ListIF <TreeIF <E>> getChildren ();
    /* Obtiene el hijo que ocupa la posición dada por parámetro. *
     * @param pos la posición del hijo que se desea obtener,     *
     * comenzando en 1.                                          *
     * @Pre 1 \leq pos \leq getChildren().size() && !isEmpty()   *
     * @return el árbol hijo que ocupa la posición pos.          */
     public TreeIF<E> getChild (int pos);
    /* Inserta un árbol como hijo en la posición pos.            *
     * @param pos la posición que ocupará el árbol entre sus     *
     * hermanos, comenzando en 1.                                *
     * Si pos == getChildren ().size () + 1, se añade como       *
     * último hijo.                                              *
     * @param e el hijo que se desea insertar.                   *
     * @Pre 1 \leq pos \leq getChildren().size()+1 && !isEmpty() */
     public void addChild (int pos, TreeIF<E> e);
    /* Elimina el hijo que ocupa la posición parámetro.          *
     * @param pos la posición del hijo con base 1.               *
     * @Pre 1 \leq pos \leq getChildren().size() && !isEmpty()   */
     public void removeChild (int pos);
    /* Determina si el árbol llamante es una hoja.               *
     * @Pre: !isEmpty (); (un arbol vacio no se considera hoja)  *
     * @return el árbol es una hoja (no tiene hijos).            */
     public boolean isLeaf ();
    /* Obtiene un iterador para el árbol.                        *
     * @param traversal el tipo de recorrido indicado por las    *
     * constantes PREORDER (preorden o profundidad), POSTORDER   *
     * (postorden) o BREADTH (anchura)                           *
     * @return un iterador según el recorrido indicado           */
     public IteratorIF<E> iterator (int traversal);
}
```

BTreeIF (Árbol Binario)

```
/* Representa un arbol binario de elementos                     */
public interface BTreeIF<E> extends CollectionIF<E>{
  public int PREORDER  = 0;
  public int POSTORDER = 1;
  public int BREADTH = 2;
```

```
  public int INORDER   = 3;
  public int RLBREADTH = 4;
  /* Obtiene la raíz del árbol (único elemento sin antecesor).    *
   * @Pre: !isEmpty ();                                           *
   * @return el elemento que ocupa la raíz del árbol.             */
  public E getRoot ();
  /* Obtiene el hijo izquierdo del árbol llamante o un árbol      *
   * vacío en caso de no existir.                                 *
   * @Pre: !isEmpty ();                                           *
   * @return un árbol, bien el hijo izquierdo bien uno vacío      *
   * de no existir tal hijo.                                      */
  public BTreeIF<E> getLeftChild ();
  /* Obtiene el hijo derecho del árbol llamante o un árbol        *
   * vacío en caso de no existir.                                 *
   * @Pre: !isEmpty ();                                           *
   * @return un árbol, bien el hijo derecho bien uno vacío        *
   * de no existir tal hijo.                                      */
  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.                         *
   * @Pre: !isEmpty ();                                           */
  public void removeLeftChild ();
  /* Elimina el hijo derecho del árbol.                           *
   * @Pre: !isEmpty ();                                           */
  public void removeRightChild ();
  /* Determina si el árbol llamante es una hoja.                  *
   * @Pre: !isEmpty (); (un arbol vacio no se considera hoja)     *
   * @return true sii el árbol es una hoja (no tiene hijos).      */
  public boolean isLeaf ();
  /* Obtiene un iterador para el árbol.                           *
   * @param traversal el tipo de recorrido indicado por las       *
   * constantes PREORDER (preorden o profundidad), POSTORDER      *
   * (postorden), BREADTH (anchura), INORDER (inorden) o          *
   * RLBREADTH (anchura de derecha a izquierda).                  *
   * @return un iterador según el recorrido indicado.            */
  public IteratorIF<E> iterator (int traversal);
}
```

ComparatorIF (Comparador)

```
/* Representa un comparador entre elementos respecto a una       *
```

```
 * relación de (al menos) preorden.                                    */
public interface ComparatorIF<E>{
  /* Sean a, b elementos de un conjunto dado y \sqsubset la           *
   * relación que establece un preorden entre ellos (nótese           *
   * que \sqsupset sería la relación recíproca, es decir, en          *
   * sentido opuesto a \sqsubset):                                    */
  public static int LT = -1; // Less than: a \sqsubset b
  public static int EQ = 0;  // Equals: !(a \sqsubset b) &&
                             // && !(a \sqsupset b)
  public static int GT = 1;  // Greater than: a \sqsupset b
  /* Compara dos elementos respecto a un preorden e indica su         *
   * relación respecto al mismo, es decir, cuál precede al            *
   * otro mediante esa relación.                                      *
   * @param a el primer elemento.                                     *
   * @param b el segundo elemento.                                    *
   * @return LT sii a \subsesq b;                                     *
   *         EQ sii !(a \subsetsq b) && !(a \sqsupset b)              *
   *         GT sii a \sqsupset b                                     */
  public int compare (E a, E b);
  /* Determina si el primer parámetro precede en el preorden          *
   * al segundo (a < b).                                              *
   * @param a el primer elemento.                                     *
   * @param b el segundo elemento.                                    *
   * @return a \sqsubset b;                                           */
  public boolean lt (E a, E b);
  /* Determina si el primer parámetro es igual al segundo en          *
   * el preorden.                                                     *
   * @param a el primer elemento.                                     *
   * @param b el segundo elemento.                                    *
   * @return a EQ b sii !(a \sqsubset b) && !(a \sqsupset b)          */
  public boolean eq (E a, E b);
  /* Determina si el primer parámetro sucede en el preorden           *
   * al segundo (b > a).                                              *
   * @param a el primer elemento.                                     *
   * @param b el segundo elemento.                                    *
   * @return a GT b sii b \sqsupset a                                 */
  public boolean gt (E a, E b);
}
```

IteratorIF (Iterador)

```
/* Representa un iterador sobre un Tipo Abstracto de Datos.          */
public interface IteratorIF<T>{
   /* Obtiene el siguiente elemento de la iteración.                 *
    * @Pre: hasNext ();                                              *
    * @return el siguiente elemento de la iteración,                 */
   public T getNext ();
   /* Comprueba si aún quedan elementos por iterar.                  *
    * @return true sii el iterador dispone de más elementos.        */
   public boolean hasNext ();
   /* Vuelve la posición del iterador al principio. Esto             *
    * permite reutilizar un iterador evitando crear otro nuevo.     */
   public void reset ();
}
```