Ejercicios de Algoritmos y Estructuras de Datos

ETSIINF - Universidad Politécnica de Madrid

Enero 2009 – Julio 2021

Profesores AED
ETSIINF — UPM

Versión de 11 de enero de 2023

Leyenda

fácil = fácil.

medio = dificultad media.

para pensar = algo complicado.

Índice

Arrays

Ejercicio 1: Implementación de add en `IndexedList`

fácil

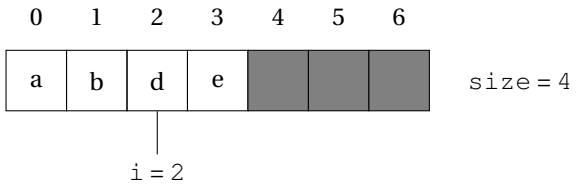
Se tiene el siguiente fragmento incompleto de la clase `ArrayIndexedList<E>`:

```
public class ArrayIndexedList<E> implements IndexedList<E> {
  private E[] A ;
  private int capacity = 6 ;
  private int size ;

  protected void checkIndex(int i, int n)
    throws IndexOutOfBoundsException {
    if (i < 0 || i >= n)
      throw new IndexOutOfBoundsException("Illegal index: " + i) ;
  }

  public void add(int i, E e) throws IndexOutOfBoundsException {
    checkIndex(i, size + 1) ;
    /** COMPLETAR **/
  }
}
```

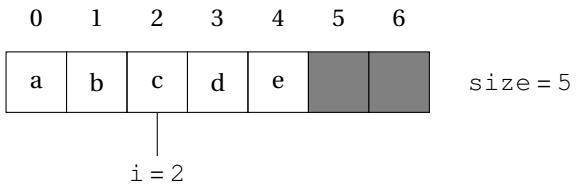
El atributo `A` es el array en el que se almacenan los elementos. El atributo `capacity` guarda el tamaño del array. El atributo `size` guarda el número de elementos almacenados en el array. El método `checkIndex` comprueba que el índice `i` está entre 0 y el valor del argumento `n`.

Completar el código Java del método `add`, el cual debe insertar el elemento `e` en la posición `i` del array (desplazando elementos a la derecha para hacer hueco si fuera necesario) y ajustar el valor de `size`. Nota: será necesario incrementar el tamaño del array en ningún caso, asumimos que `size < A.length`.

Por ejemplo, cuando el array contiene los siguientes 4 elementos y se invoca `add(2,c)`:



El método debe desplazar a la derecha los elementos en el array desde la posición 2 en adelante e insertar el nuevo elemento en dicha posición:



Ejercicio 2: Implementación de `remove` en `IndexedList`

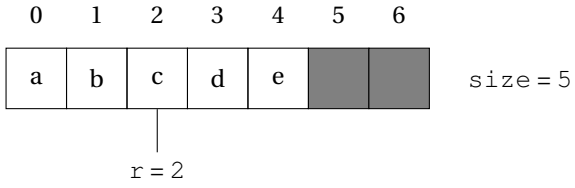
fácil

Dado el siguiente fragmento incompleto de la clase `ArrayIndexedList<E>`:

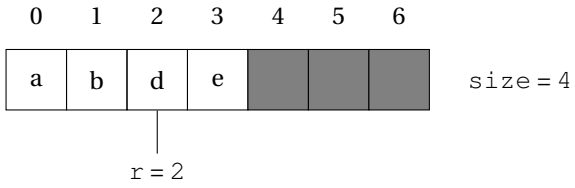
```
public class ArrayIndexedList<E> implements IndexedList<E> {
  private E[] A;
  private int capacity = 6;
  private int size;

  ...

  /** Checks whether the given index is in the range [0, n-1] */
  protected void checkIndex(int r, int n)
    throws IndexOutOfBoundsException {
    if (r < 0 || r >= n)
      throw new IndexOutOfBoundsException("Illegal index: " + r);
  }

  ...

  public E remove(int r) throws IndexOutOfBoundsException {
    checkIndex(r, size());
    E temp = A[r];

    // COMPLETAR //

    return temp;
  }
}
```

donde `A` es el *array* en el que se almacenan los elementos, `capacity` guarda el tamaño del *array* y `size` guarda el número de elementos almacenados en el *array*.

Completar el método `remove` con el código que elimina el elemento `A[r]`, desplaza los elementos con índices que `r` una posición a la izquierda y ajusta el valor de `size`. Por ejemplo, cuando el *array* contiene los siguientes 5 elementos y se invoca `remove(2)`:



El método debe eliminar el elemento en el índice 2 y mover los elementos siguientes una posición a la izquierda:



Ejercicio 3: Teoría sobre arrays

fácil

Queremos implementar una lista utilizando *arrays* de Java. En un momento cualquiera, `n` es el número de elementos que hay en la lista. Di cuál de las siguientes opciones es verdadera:

(a) `n` no puede ser mayor que el mayor índice válido del *array*.

(b) `n` siempre tiene que ser menor que el mayor índice válido del *array*.

(c) `n` siempre tiene que ser menor o igual que la longitud del *array*.

(d) `n` no puede ser igual a la longitud del *array*.

Ejercicio 4: Método `reverse` de un array

fácil

Implementar en Java el método

```
public int [] reverse(int [] v)
```

que devuelve un nuevo array de enteros que contendrá los elementos de `v` pero en orden inverso. Por ejemplo, si `v` referencia un array con los elementos 1,4,6,3, entonces el método `reverse` debe devolver un nuevo array con los elementos 3,6,4,1.

Ejercicio 5: Método `iguales`

fácil

Implementar en Java el método

```
public static boolean iguales (Integer [] arr1, Integer [] arr2)
```

que toma como parámetro dos «arrays» `arr1` y `arr2` de objetos de clase `Integer` e indica si los «arrays» son de la misma longitud y tienen los mismos elementos de izquierda a derecha. Se asume que `arr1` y `arr2` no son null pero pueden contener elementos null.

Ejercicio 6: Método `esInverso`

fácil

Implementar en Java el método:

```
public static <E> boolean esInverso (E [] arr1, E [] arr2)
```

que toma como parámetro dos «arrays» `arr1` y `arr2` de objetos de tipo genérico e indica si los «arrays» son de la misma longitud y tienen los mismos elementos pero en orden inverso. Se asume que `arr1` y `arr2` no son **`null`** pero pueden contener elementos **`null`**.

Por ejemplo, si el array `arr1` está formado por los elementos [1,2,3,4,5] y `arr2` por los elementos [5,4,3,2,1], el método `esInverso` devolverá **`true`**. Si `arr1` contiene los elementos [1,null,3,null,5] y `arr2` [5,null,3,null,1], `esInverso` también devolverá true, pero si `arr1` contiene [1,null,3,2,5] y `arr2` [5,null,3,null,1], entonces devolverá **`false`**. Para el caso [2,3,4] y [4,3,1], `esInverso` debe devolver false.

TADs

Ejercicio 7: Representación de colores

fácil

Se tiene el siguiente interfaz `ColorRGB`:

```
public interface ColorRGB {
 public int getRed();    // Devuelve el nivel de rojo del color
 public int getGreen();  // Devuelve el nivel de verde del color
 public int getBlue();   // Devuelve el nivel de azul del color
 public String getHex(); // Devuelve la representacion hexadecimal del color
}
```

Se tiene también el código incompleto de las clases `ColorHex` y `ColorInts` que implementan dicho interfaz.

```
public class ColorHex implements ColorRGB {
  private String hex;

  // Los constructores y metodos auxiliares ya estan implementados...

  public int getRed() { /* COMPLETAR */ }
  public int getGreen() { /* COMPLETAR */ }
  public int getBlue() { /* COMPLETAR */ }
  public String getHex() { /* COMPLETAR */ }
  public boolean equals(Object o) { /* COMPLETAR */ }
}

public class ColorInts implements ColorRGB {
  private int r, g, b;

  // Los constructores y motodos auxiliares ya estan implementados...

  public int getRed() { /* COMPLETAR */ }
  public int getGreen() { /* COMPLETAR */ }
  public int getBlue() { /* COMPLETAR */ }
  public String getHex() { /* COMPLETAR */ }
  public boolean equals(Object o) { /* COMPLETAR */ }
```

La clase `ColorHex` utiliza un atributo de tipo `String` para representar el color como un valor haxedecimal. La clase `ColorInts` utiliza tres atributos entero para representar los niveles de rojo, verde y azul del color. Asumimos que ya están implementados todos los constructores y los siguientes métodos auxiliares:

```
private int getRedFromHex(String hex);  // obtiene el rojo de un hexadec.
private int getBlueFromHex(String hex); // obtiene el azul de un hexadec.
private int getGreenFromHex(String hex);// obtiene el verde de un hexadec.
private String getHexValue(int r, int g, int b); // rgb a hexadec.
```

Los tres primeros permiten obtener de un hexadecimal el nivel de rojo, azul y verde. El último permite obtener el valor hexadecimal a partir de los niveles de rojo, azul y verde.

Completar los «getters» y el método `equals` de cada clase. El método `equals` debe implementarse de forma que un objeto de clase `ColorHex` (idem `ColorInts`) indique si es igual o no a cualquier objeto de cualquier

clase que implemente el interfaz `ColorRGB`. Es decir, el método `equals` debe indicar si dos colores son iguales independientemente de la representación utilizada.

Ejercicio 8: Representación la hora

fácil

Se tiene el interfaz `Time`, similar al utilizado en uno de los laboratorios de la asignatura:

```
public interface Time {
  public int getHours();
  public int getMins();
  public int getSecs();
  public int timeInSeconds();
}
```

Se tiene también el código incompleto de las clases `TimeHMS` y `TimeSec` que implementan dicho interfaz.

```
public class TimeHMS implements Time {

  private int hours;
  private int mins;
  private int secs;
  ...

  public int getHours() { /* COMPLETAR */ }
  public int getMins() { /* COMPLETAR */ }
  public int getSecs() { /* COMPLETAR */ }
  public int timeInSeconds(){ /* COMPLETAR */ }
}

public class TimeSec implements Time {

  private int totalSecs;
  ...

  public int getHour() { /* COMPLETAR */ }
  public int getMins() { /* COMPLETAR */ }
  public int getSecs() { /* COMPLETAR */ }
  public int timeInSeconds(){ /* COMPLETAR */ }
  public boolean equals (Object o) { /* COMPLETAR */ }
}
```

Al igual que en el citado laboratorio, la clase `TimeHMS` representa la hora del día mediante tres atributos, `hours`, `mins` y `sec`, mientras que la clase `TimeSec` representa la hora mediante un único atributo `totalSecs` que almacena la hora en los segundos totales transcurridos desde el principio del día. No se consideran valores negativos en ninguno de los atributos.

Completar los «getters» de las clases `TimeHMS` y `TimeSec`, así como el método `equals` de la clase `TimeSec`. El método `equals` debe implementarse de forma que un objeto de clase `TimeSec` indique si es igual o no a cualquier objeto de cualquier clase que implemente el interfaz `Time`, es decir, el método `equals` debe indicar si dos horas son iguales independientemente de la representación utilizada.

Comparadores y Ordenación

Ejercicio 9: Implementación de comparadores

medio

Se desea implementar una clase de objetos comparadores que comparen objetos iteradores:

```
public class IteratorComparator<E> implements Comparator<Iterator<E>> {
  private Comparator<E> compElem;

  public IteratorComparator(Comparator<E> compElem) {
    this.compElem = compElem;
  }

  public int compare(Iterator<E> it1, Iterator<E> it2) {
    // COMPLETAR
  }
}
```

El constructor de la clase toma como parámetro un comparador de elementos `compElem`. Implementar el método `compare` que debe devolver el resultado de comparar los iteradores `it1` e `it2` que toma como parámetros. La comparación de dichos iteradores se realiza comparando los elementos que devuelven `it1.next()` e `it2.next()` con el comparador de elementos. En el momento en el que la comparación de elementos no sea 0, el resultado de la comparación de iteradores será el de dicha comparación de elementos. Si todas las comparaciones de elementos devuelven 0, se considera «mayor» el iterador al que le quedan elementos por recorrer.

Ejercicio 10: Ordenar datos

medio

Se tiene la siguiente clase `Dato` que implementa el interfaz `Comparable<Dato>`:

```
public class Dato implements Comparable<Dato> {
  private int valor;

  public Dato(int valor) { this.valor = valor; }

  /* FALTA CODIGO */
}
```

Esta clase contiene un atributo `valor` que contiene un número y que nos permitirá comparar dos elementos de tipo `Dato`. Se considera que un objeto de tipo `Dato` es mayor cuanto mayor sea el valor almacenado en el atributo `valor`.

(a) Implementar el/los métodos necesarios para que la clase implemente el interfaz `Comparable<Dato>`.

(b) Implementar el método `estaOrdenado`:

```
public static boolean estaOrdenado (Dato [] datos)
```

que recibe un array de objetos de tipo `Dato` y devuelve **`true`** si éste está ordenado de forma ascendente o **`false`** en caso contrario. El array `datos` puede contener repetidos. Dicho método se implementa en una clase desde la que no está accesible el atributo `valor`, con lo que no se puede acceder directamente a dicho atributo en la implementación de `estaOrdenado`. El parámetro `datos` no será **`null`** y no contendrá elementos **`null`**.

Ejercicio 11: Método `estaOrdenada`

medio

Implementar en Java el método:

```
static <E extends Comparable<E>> boolean estaOrdenada (PositionList<E> list)
```

que recibe como parámetro una lista cuyos elementos implementan el interfaz `Comparable` y devuelve **`true`** si ésta está ordenada en orden ascendiente y **`false`** en caso contrario. La lista de entrada no será **`null`** y la lista no contendrá elementos **`null`**. La expresión `<E extends Comparable<E>>` indica que los objetos de la clase `E` implementan el interfaz `Comparable<E>`.

Por ejemplo, dada las listas de entrada [1,2,2,3,5], [1] o [] el resultado será **`true`**. Dada la lista [4,1,2,3,4] el resultado será **`false`**.

Uso de Listas

Ejercicio 12: Sobre iterador de listas

para pensar

A continuación está el código de la clase `IteradorListas` que implementa un iterador para PositionList<E>.

```
public class IteradorListas<E> implements Iterator<E> {
  protected PositionList<E> list; // la lista sobre la que se itera
  protected Position<E> cursor; // la  siguiente posición en el recorrido

  /** Crea un objeto iterador para la lista pasada como argumento. */
  public IteradorListas(PositionList<E> L) {
     list = L;
     cursor = (list.isEmpty())? null : list.last();
  }

  public boolean hasNext() { return (cursor != null);  }

   public E next() throws NoSuchElementException {
     if (cursor == null)
        throw new NoSuchElementException("No hay siguiente elemento");
     E elem = cursor.element();
     cursor = (cursor == list.first())? null : list.prev(cursor);
     return elem;
   }

   public void remove() throws UnsupportedOperationException {
     throw new UnsupportedOperationException("remove");
   }
}
```

Sobre él, responde a las siguientes preguntas:

(a) Queremos que un iterador de esa clase recorra las listas empezando en el elemento almacenado en la primera posición y avanzando hacia el último. Decide si con el código actual ésto es así, y si crees que no, dí qué cambios habría que hacer en el código de los métodos de la clase.

1. ¿Cambios en el Constructor IteradorListas?:
2. ¿Cambios en el método hasNext?:
3. ¿Cambios en el método next?:

(b) Utilizar iteradores de esa clase para implementar el método iguales, que devuelve true si las dos listas que recibe como argumento son iguales (mismo tamaño, contienen los mismos elementos en el mismo orden) y false en caso contrario.

(c) Di cuál es la complejidad del algoritmo que has escrito, razonándolo brevemente.

Ejercicio 13: `take` con `IndexedList`

fácil

Completar el código Java del siguiente método:

```
public IndexedList<E> take(int n, IndexedList<E> list) {
  IndexedList<E> res = new ArrayIndexedList<E>() ; // lista resultado
  /** COMPLETAR **/
}
```

El método toma un entero `n` y una lista indexada `list` y devuelve una nueva lista indexada con los `n` primeros elementos (en el mismo orden) de `list`.

Cuando `n`≤ 0 la lista devuelta es vacía. Cuando `n`≥`list.size()` la lista devuelta tiene todos los elementos (en el mismo orden) de `list`.

Ejercicio 14: `take` con `PositionList`

fácil

Se ha decidido añadir el siguiente método al intefaz `PositionList<E>`:

```
public PositionList<E> take(int n, PositionList<E> list);
```

el cual dado un entero `n` y una lista `list` devuelve una nueva lista con los `n` primeros elementos de `list` en el mismo orden en el que aparecen en ésta. Cuando `n` es menor o igual a 0 entonces la lista devuelta es vacía. Cuando `n` es mayor o igual al número de elementos entonces la lista devuelta es idéntica a `list`.

Por ejemplo, dada la lista [1,3,5,7,4,0], el método `take` con `n` igual a 4 sobre dicha lista devuelve la nueva lista [1,3,5,7]. Con `n` igual a 8 sobre [1,3,5,7,4,0] devuelve la nueva lista [1,3,5,7,4,0].

Como es de esperar, debe añadirse la implementación de dicho método a la clase `NodePositionList<E>` que implementa `PositionList<E>`. Mostramos a continuación el fragmento incompleto de dicha implementación:

```
public class NodePositionList<E> implements PositionList<E> {
  ...
  public PositionList<E> take(int n, PositionList<E> list) {
    PositionList<E> rlist = new NodePositionList<E>(); // lista resultado vacia

    // COMPLETAR //

    return rlist;
  }
}
```

Completar el código del método. Dicho código debe usar un iterador sobre las posiciones devueltas por `list.positions()`, almacenando en la lista resultado `rlist` los `n` primeros elementos en las posiciones devueltas por el iterador.

Ejercicio 15: Suma de elementos de una lista

fácil

Dada el siguiente método:

```
public int suma (PositionList<Integer> lista);
```

escribir en Java el método `suma` que devuelva la suma de los elementos que componen la lista.

Ejercicio 16: Gestión de pedidos

fácil

Se tiene una aplicación de gestión de pedidos que necesita almacenarlos en el orden inverso en que han sido realizados. Se decide usar un TAD lista para almacenar los pedidos. Cuando un cliente realiza un pedido p_1 y después otro pedido p_2 éstos se almacenan en la lista en orden inverso: p_2, p_1. Se dispone únicamente de las implementaciones `ArrayIndexedList` y `NodePositionList`. Razonar cuál de las dos implementaciones sería la idónea en términos de mejor complejidad de almacenamiento.

Ejercicio 17: Método `append`

fácil

Se desea añadir el siguiente método al interfaz `PositionList<E>`:

```
public void append(List<E> list)
```

El método debe «concatenar» la lista que se pasa como parámetro a la lista sobre la que se invoca el método. Dada la invocación `list1.append(list2)` el método debe añadir en orden de aparición los elementos del objeto `list2` al final del objeto `list1`. Por ejemplo, si `list1` contiene los elementos 1,3,4 y `list2` los elementos 7,8, después de la invocación `list1` contendrá los elementos 1,3,4,7,8. Obsérvese que `append` **no** lanza excepciones. Escribir la implementación del método utilizando únicamente los métodos ofrecidos por el interfaz `PositionList<E>`.

Ejercicio 18: Método `addBeforeElement`

fácil

Implementar en Java el siguiente método:

```
public void addBeforeElement(PositionList<E> list, E e1, E e2)
```

que inserta en una lista de posiciones dada `list` el elemento `e1` en el nodo (`Position`) justamente anterior al que almacena el elemento `e2`, si este último está en la lista. Si `e2` no está en la lista el método deja la lista intacta.

Ejercicio 19: Método `find`

fácil

Se desea implementar un método `find` que toma como parámetros una `PositionList` así como un elemento y debe devolver el objeto posición donde se encuentra la primera ocurrencia del elemento en la lista. En caso de que el elemento no esté en la lista el método debe devolver **`null`**. A continuación se muestra una implementación incorrecta:

```
public Position<E> find(PositionList<E> list, E e) {
  for ( Position<E> p = list.first();
        !p.element().equals(e);
        p = list.next(p)
      ) ;
  return p.element().equals(e) ? p : null;
}
```

Indicar brevemente dos errores en dicha implementación que pueden ser causa de excepciones.

Ejercicio 20: Método `replace` fácil

Implementar en Java el método:

```
public void replace(IndexedList<E> list, E e1, E e2)
```

que reemplaza todas las ocurrencias del elemento `e1` en la lista indexada `list` por el elemento `e2`.

Ejercicio 21: Método `reverse` fácil

Implementar en Java el método:

```
public void reverse(ArrayIndexedList<E> list)
```

que invierte la lista que toma como parámetro. Por ejemplo, si la lista contiene los elementos 2,1,5,4 en ese orden, entonces el método debe devolver una lista con los elementos 4,5,1,2. El método debe invertir la lista con una buena complejidad. En particular se valorarán más las soluciones que inviertan la lista en $O(n)$.

Ejercicio 22: Copiar lista hasta `null` fácil

Implementar en Java el método

```
public PositionList<E> hastaNull(PositionList<E> list)
```

que toma como parámetro una lista, `list`, de elementos algunos de los cuales podrían ser **`null`**. El método debe devolver una nueva lista con los elementos de `list` en el mismo orden de aparición de izquierda a derecha, pero parando al encontrar el primer elemento **`null`** en `list`. Si no hay elementos **`null`** entonces la nueva lista debe tener todos los elementos de `list` en el mismo orden. Por ejemplo, dada la lista con los elementos 1, 4, null, 3, null, 7, el método debe devolver una nueva lista con los elementos 1, 4. Dada la lista con los elementos 1,4,3,7, el método debe devolver una nueva lista con 1,4,3,7.

Ejercicio 23: Contar apariciones en una lista fácil

Implementar en Java el método

```
public int numoc(E elem, PositionList<E> list)
```

que devuelve el número de ocurrencias del elemento `elem` en la lista `list`. El método debe implementarse utilizando el iterador de la lista `list`.

Ejercicio 24: ¿Son dos listas iguales? medio

Se decide añadir el método `equals` a la clase `NodePositionList<E>` la cual implementa el interfaz `PositionList<E>`:

```
public boolean equals(PositionList<E> list) {
  if (this == list) return true ;
  // COMPLETAR
}
```

Completar el código de `equals` sabiendo que dos listas son iguales si las dos son vacías o las dos contienen los mismos elementos de izquierda a derecha.

Ejercicio 25: Eliminar un entero de una lista

fácil

Implementar el método

```
public void filter(int i, PositionList<Integer> list)
```

que debe borrar de la lista `list` todos los objetos `Integer` que contengan el mismo valor que `i`.

Ejercicio 26: Completar `addFirst()`

fácil

Escribir el código del método `addFirst` de la clase `NodePositionList<E>`. A continuación recordamos las primeras líneas de código de dicha clase en las que se muestran sus tres atributos:

```
public class NodePositionList<E> implements PositionList<E> {
  protected int numElts;              // Number of elements in the list
  protected Node<E> header, trailer;  // Special sentinels
  ...
  /** Inserts an element at the front of the list, creating new position. */
  public void addFirst(E element) {
    /* COMPLETAR */
  }
```

Ejercicio 27: Eliminar apariciones de un elemento en una lista

fácil

Implementar en Java el método:

```
public void removeAll(PositionList<E> list, E e)
```

Dicho método debe eliminar todas las ocurrencias del objeto elemento `e` de la lista `list`. La lista debe quedar intacta si no hay ninguna ocurrencia del elemento.

Ejercicio 28: Eliminar elementos pertenecientes a otra lista

medio

Supongamos que dentro de una clase se ha implementado el siguiente método:

```
public boolean member(E elem, PositionList<E> list) { ... }
```

El método `member` indica si el elemento `elem` está en la lista `list`.

Implementar el método:

```
public void removeList(PositionList<E> l1, PositionList<E> l2)
```

Dicho método estará en la misma clase que `member`. El método toma como parámetro dos listas `l1` y `l2` ambas distintas de **`null`**, ambas sin elementos **`null`**, y ambas sin elementos repetidos. El método debe modificar la lista `l1` borrando de `l1` aquellos elementos que estén en la lista `l2`. Por ejemplo, si `l1` contiene los elementos 1,5,3,7,8 y `l2` contiene los elementos 5,6,1,8, tras la invocación de `removeList` la lista `l1` debe contener los elementos 3,7.

Ejercicio 29: Utilidad de `CheckPosition`

fácil

En la clase `NodePositionList<E>` se utiliza el siguiente método protegido:

```
protected Node<E> checkPosition(Position<E> p)
  throws InvalidPositionException
```

(a) Indicar qué métodos de la clase `NodePositionList<E>` invocan dicho método (no es necesario listarlos, sólo indicar qué tienen en común).

(b) Describir brevemente cuál es el resultado de `checkPosition` y bajo qué condiciones lanza una excepción.

Ejercicio 30: Aplanar una lista

medio

Implementar en Java el método:

```
public static PositionList<Integer>
  aplanar (PositionList<PositionList<Integer>> listaDeListas)
```

que toma como parámetro una lista cuyos elementos son a su vez listas de enteros. El método debe devolver una nueva lista de enteros con todos los enteros en el orden en el que aparecen en la lista de listas. Por ejemplo, dada la lista de listas [[1,3], [], [7,6,1], [], [9,0], [3]] el método debe devolver la lista [1,3,7,6,1,9,0,3]. En este ejemplo estamos usando corchetes para denotar una lista, por tanto [] es la lista vacía. Puede asumirse que ninguna lista o elemento es **`null`**. La solución debe usar iteradores explícitamente o mediante «for-each».

Ejercicio 31: Eliminar elementos entre dos posiciones

medio

Se desea implementar en Java el método

```
public static <E> void delete( PositionList<E> list,
                               Position<E> pos1,
                               Position<E> pos2 )
```

que toma como parámetro una lista de posiciones `list`, que asumimos que no es null y no es vacía, y dos posiciones `pos1` y `pos2` que referencian un nodo válido de dicha lista (bien el mismo nodo o nodos diferentes). El método debe determinar primero si el nodo que referencia `pos1` está a la izquierda del que referencia `pos2` o vice versa. Una vez determinado ésto, debe borrar todos los nodos entre `pos1` y `pos2`, ámbos inclusive, cuando `pos1` está a la izquierda de `pos2`, o alternativamente, todos los nodos entre `pos2` y `pos1`, ámbos inclusive, cuando `pos2` está a la izquierda de `pos1`. Si `pos1` y `pos2` referencian el mismo nodo entonces sólo borra ese nodo.

Ejercicio 32: Devolver un elemento en una posición de una lista

fácil

Implementar en Java el método

```
public static <E>  E get(PositionList<E> list, int n)
```

que toma como parámetro una lista de posiciones `list` y un índice entero `n`, y devuelve el elemento en el `n`-ésimo nodo de la lista cuando dicho nodo `n`-ésimo existe, o **`null`** en caso contrario. El nodo `n`-ésimo existe cuando `n` está entre 0 y `list.size()-1`.

Ejercicio 33: Error añadiendo un elemento a una lista

fácil

Dado el siguiente código incorrecto:

```
PositionList<String> list1 = new NodePositionList<String>();
PositionList<String> list2 = new NodePositionList<String>();
list1.addFirst("mundo");
list2.addFirst("mundo");
list1.addBefore(list2.first(),"hola");
list2.addBefore(list2.first(),"hola");
```

Indique qué línea es incorrecta y explique brevemente el error que se producirá.

Ejercicio 34: Método `flatNub`

medio

Implementar en Java el método:

```
public static PositionList<Integer> flatNub (PositionList<Integer> [] arr)
```

que toma como parámetro un «array» `arr` que no será null y cuyos elementos no serán null sino que referenciarán objetos listas de posiciones cuyos elementos tampoco serán null sino que referenciarán objetos de tipo `Integer`. Si `arr` es vacío entonces el método debe devolver una nueva lista vacía. En otro caso, el método debe recorrer el «array» de izquierda a derecha y por cada lista del mismo recorrer ésta de izquierda a derecha insertando al final de la lista resultado los elementos que no estén ya en la lista resultado.

Por ejemplo, sea `arr` un «array» de tamaño 3 donde `arr[0]` referencia la lista [0,2,5,1], `arr[1]` referencia la lista [1,3,7,2] y `arr[2]` referencia la lista [4,1,2,5]. La lista resultado debe ser [0,2,5,1,3,7,4]. Puede asumirse que se dispone de la clase `NodePositionList<E>` que implementa listas de posiciones, y que también se dispone del método auxiliar

```
private static <E> boolean member(E elem, PositionList<E> list)
```

que indica si un elemento `elem` aparece en la lista `list`.

Ejercicio 35: Método `join`

medio

Implementar en Java el método:

```
public static <E> PositionList<Integer> join (PositionList<Integer> l1,
                                              PositionList<Integer> l2)
```

que toma como parámetro dos listas de tipo `PositionList<Integer>`. Los elementos de ambas listas *siempre estarán ordenados en orden creciente* y el método `join` debe devolver una *nueva* lista ordenada que contenga los elementos de las listas `l1` y `l2`. Las listas `l1` y `l2` no contendrán elementos repetidos, aunque podrá haber elementos que estén contenidos en ambas listas. La longitud de la lista resultado será la misma que la suma de las longitudes `l1` y de `l2`, esto implica que, los elementos que estén en ambas listas, deben ser insertados dos veces en la lista resultante. Se asume que `l1` y `l2` no son null y que tampoco podrán contener elementos **`null`**.

Por ejemplo, si `l1` es la lista formada por los elementos [1,2,4,5,6] y `l2` por los elementos [1,2,3], el método `join` devolverá una nueva lista con los elementos [1,1,2,2,3,4,5,6]. Si `l1` es [2] y `l2` es [1,3,5,6], el método `join` devolverá [1,2,3,5,6].

Ejercicio 36: Método `intercalar`

medio

Implementar en Java el método:

```
public static <E> PositionList<Integer> intercalar(PositionList<Integer> l1,
                                                   PositionList<Integer> l2)
```

que toma como parámetro dos listas de tipo `PositionList<Integer>`. El método debe devolver una *nueva* lista que contenga los elementos de las listas `l1` y `l2` intercalándolos de forma alterna y empezando por los elementos de `l1`. Si `l1` y `l2` tienen distinto tamaño, se añadirán los elementos «sobrantes» de la lista más larga al final de la lista resultado. La longitud de la lista resultado será la misma que la suma de las longitudes `l1` y de `l2`. Se asume que `l1` y `l2` no son null, pero pueden estar vacías.

Por ejemplo, si `l1` es la lista formada por los elementos [1,2,3,4] y `l2` por los elementos [5,6,7,8], el método `intercalar` devolverá una nueva lista con los elementos [1,5,2,6,3,7,4,8]. Si `l1` es [6,1] y `l2` es [3,4,8,7], el método `intercalar` devolverá [6,3,1,4,8,7]. O si `l1` es [3,4,5,6] y `l2` es [], el método `intercalar` devolverá [3,4,5,6].

Ejercicio 37: Método `getNumApariciones`

medio

Implementar en Java **de forma recursiva** el método

```
static <E> int getNumApariciones (PositionList<E> lista, E elem)
```

que devuelve el numero de apariciones de `elem` en `lista`. El uso de bucles (while, for o do-while) y de iteradores NO está permitido. La `lista` podrá ser **`null`**, en cuyo caso el método devolverá 0 apariciones. La lista no contendrá elementos **`null`** y el parámetro `elem` nunca será **`null`**.

Ejercicio 81: Iterador Positivos

fácil

Dada la siguiente implementación del iterador `PositivePositionListIterator`.

```
public class PositivePositionListIterator<E extends Integer>
                                  implements Iterator<Integer> {

  private PositionList<Integer> list;
  private Position<Integer> cursor;

  public PositivePositionListIterator(PositionList<Integer> list) {
    if (list == null) {
      throw new IllegalArgumentException ("list no puede ser null");
    }
    this.list   = list;
    this.cursor = list.first();
    moveCursor(); /*AQUI*/
  }

  public boolean hasNext() {
    return cursor != null;
  }

  public Integer next() throws NoSuchElementException {
    if (cursor == null) {
      throw new NoSuchElementException();
```

```
    }

    Integer e = cursor.element();
    cursor = list.next(cursor);
    moveCursor(); /*AQUI*/
    return e;
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }

  private void moveCursor() {
     /* IMPLEMENTAR */
  }
}
```

Dicho iterador permite iterar sobre una lista de tipo `PositionList<Integer>` devolviendo únicamente aquellos elementos de la lista que sean mayores que 0. La lista a iterar podrá contener elementos **`null`**. Por ejemplo, dada la lista [-1,0,1,3,-2,7,null] el iterador devolverá los elementos 1,3,7 en este orden, o dada la lista [null,-1,0] el iterador no devolverá ningún elemento. En la implementación dada, el cursor siempre debe estar apuntando al siguiente elemento a devolver por el iterador o a **`null`** en caso de que no queden elementos por devolver, siendo el método `moveCursor` el encargado de darle el valor adecuado al atributo `cursor`.

Implementar el método **`private void`** `moveCursor()`, invocado en las líneas marcadas con `/*AQUI*/`, que avanza el atributo `cursor` hasta el siguiente elemento a devolver por el iterador, o lo deja a **`null`** si no hay mas elementos que devolver.

Ejercicio 39: Método `getElementosImpares`

fácil

Implementar en Java el método:

```
PositionList<Integer> getElementosImpares (PositionList<Integer> lista)
```

que recibe como parámetro una lista cuyos elementos pueden ser null y devuelve una *nueva* lista donde únicamente se incluyen los elementos impares contenidos en `lista`. La lista resultante no contendrá elementos repetidos y el orden de los elementos en la lista resultante no es relevante. Si `lista` es **`null`** el método debe lanzar la excepción `IllegalArgumentException`.

Por ejemplo, dada la siguiente lista de entrada [1,2,null,1,4,7,8] el resultado podría ser [1,7] o [7,1]. Dada la lista [null,1,7,2,9,null] el resultado podría ser [1,7,9], [9,7,1], [1,9,7],.... Puede asumirse que se dispone de la clase `NodePositionList<E>` que implementa listas de posiciones, y que también se dispone del método auxiliar

```
private static <E> boolean member(E elem, PositionList<E> lista)
```

que indica si un elemento `elem` está contenido en la lista `lista`.

Ejercicio 40: Método `eliminarConsecutivosIguales`

fácil

Implementar en Java el método:

```
static <E> void eliminarConsecIguales (PositionList<E> lista)
```

que recibe como parámetro una lista de posiciones `lista` y elimina de dicha lista los elementos consecutivos que sean iguales. La lista no será **`null`**, pero los elementos contenidos en la lista sí pueden ser **`null`**. Por ejemplo, dada `lista=[5,1,1,2,3,3,5,null,null,1]`, la llamada a `eliminarConsecIguales(lista)` dejará en `lista` los elementos `[5,1,2,3,5,null,1]`.

Ejercicio 41: Método `barajar`

fácil

Implementar en Java el método:

```
static <E> PositionList<E> barajar (PositionList<E> list)
```

que recibe como parámetro una lista `list` y devuelve una **nueva** lista que contiene los elementos contenidos en `list` en el siguiente orden: primero, último, segundo, penúltimo, tercero, antepenúltimo,... y así hasta completar todos los elementos de la lista. La lista `list` podrá ser **`null`**, en cuyo caso el método deberá lanzar la excepción `IllegalArgumentException`. En caso de no ser **`null`**, la lista siempre contendrá al menos un elemento, y su tamaño siempre será un número impar, es decir, siempre se cumplirá `list.size()%2 == 1`.

Por ejemplo, dado `list = [1,2,3,8,4,5,6]`, la llamada `barajar(list)` devolverá `[1,6,2,5,3,4,8]`, si `list = [1,2,null,7,4,5,null]` devolverá `[1,null,2,5,null,4,7]`. Para crear la lista, se dispone de la clase `NodePositionList<E>` que implementa el interfaz `PositionList<E>` y que tiene un constructor sin parámetros.

Ejercicio 42: Método `copiarHastaElem`

fácil

Implementar en Java el método:

```
static <E> PositionList<E> copiarHastaElem(PositionList<E> list, E elem)
```

que recibe como parámetro una lista `list` y un elemento `elem`. El método debe devolver una **nueva** lista que copie los elementos contenidos en `list`, en el mismo orden, hasta la primera aparición de `elem` en `list`, sin incluir el elemento buscado. En caso de que `elem` no esté en la lista el método debe devolver una copia de la lista `list`. El parámetro `list` no será **`null`** pero podrá contener elementos **`null`**. El parámetro `elem` podrá ser **`null`**. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, dado `list = [1,2,3,4,null,6]`, la llamada `copiarHastaElem(list,4)` devolverá `[1,2,3]`, `copiarHastaElem(list,null)` devolverá `[1,2,3,4]` o `copiarHastaElem(list,14)` devolverá `[1,2,3,4,null,6]`.

Ejercicio 43: Método `invertir` recursivo

fácil

Implementar de forma **recursiva** en Java el método:

```
static <E>  PositionList<E> invertir (PositionList<E> list)
```

que recibe como parámetro una `PositionList` devuelve una **nueva** lista con los mismos elementos en orden inverso. La lista `list` podrá ser **`null`**, en cuyo caso, el método debe devolver **`null`**. El uso de bucles (while, for o do-while, for-each) así como de iteradores NO está permitido, se debe implementar mediante un **método auxiliar** que sea **recursivo**. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, dada la lista `list = [1,2,3,4,5]`, el método devolverá una **nueva** lista que contendrá `[5,4,3,2,1]`.

Ejercicio 44: Método `invertir` (inline)

fácil

Implementar en Java el método:

```
static <E> void invertir (PositionList<E> list)
```

que recibe como parámetro una lista `list` e invierte **el orden de los elementos contenidos en la lista** `list` (no se debe crear una nueva lista). La lista `list` podrá ser **`null`**, en cuyo caso el método deberá lanzar la excepción `IllegalArgumentException`.

Por ejemplo, dado `list = [1,2,3,8,4,5,6]`, la llamada `invertir(list)` dejará la lista `list = [6,5,4,8,3,2,1]`.

Ejercicio 45: Método `copyElemsNbyN`

fácil

Implementar en Java el método:

```
static <E> PositionList<E> copyElemsNbyN (PositionList<E> list, int n)
```

que recibe como parámetro una lista `list` y un número `n`. El método debe devolver una **nueva** lista que copie los elementos contenidos en `list` saltando de `n` en `n`, es decir, se deben añadir a la lista resultante los elementos que ocupan las posiciones $0, n, 2*n, 3*n, \ldots$ Nótese que el primer elemento de la lista ocupa la posición 0 y el último ocupa la posición `list.size()-1`. La lista `list` nunca será **`null`**. En caso de que `n` sea un número ≤ 0 el método deberá lanzar la excepción `IllegalArgumentException`. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, dado `list = [a,b,c,d,e,f,g,h]`, la llamada `copyElemsNbyN(list,2)` devolverá `[a,c,e,g]`, `copyElemsNbyN(list,3)` devolverá `[a,d,g]` o `copyElemsNbyN(list,5)` devolverá `[a,f]`.

Ejercicio 46: Método `average` (recursivo)

fácil

Implementar de forma **recursiva** en Java el método:

```
static double average (PositionList<Integer> list)
```

que recibe como parámetro una `PositionList` y debe devolver **la media aritmética** de los elementos contenidos en la lista `list`. El uso de bucles (**`while`**, **`for`**, **`do-while`** o **`for-each`**), así como de iteradores, NO está permitido. La implementación debe realizarse mediante un **método auxiliar** que sea **recursivo**. El argumento `list` podrá ser **`null`** o ser una lista vacía, casos en lo que el método debe lanzar la excepción `IllegalArgumentException`. `list` no contendrá elementos **`null`**.

Por ejemplo, si se invoca `average(list)` sobre la lista `list = [1,2,3]`, el método deberá devolver `2.0`; si `list = [5,2]`, deberá devolver `3.5`; y si `list = [5,4,8,2]` deberá devolver `4.75`.

Ejercicio 47: Método `quitarIguales`

fácil

Implementar en Java el método:

```
static <E> void quitarIguales (PositionList<E> list, E elem)
```

que recibe como argumentos `list` (una `PositionList`) y el elemento `elem`, y elimina todos aquellos elementos de `list` que sean iguales a `elem` sin alterar el orden relativo de los demás. Si `list` es **`null`**, el método debe lanzar la excepción `IllegalArgumentException`. El parámetro `elem` podrá ser **`null`** y la lista `list` podrá contener elementos **`null`**.

Por ejemplo, dada la lista `list = [1,2,3,null,5,3]`, la llamada a `quitarIguales(list,3)` debe dejar `list = [1,2,null,5]`; la llamada `quitarIguales(list,8)` no debe cambiar `list`; la llamada `quitarIguales(list,null)` debe dejar `list = [1,2,3,5,3]`.

Ejercicio 48: Método `countApariciones` (recursivo)

fácil

Implementar **de forma recursiva** en Java el método:

```
static <E> int countApariciones(PositionList<E> list, E elem)
```

que recibe como argumento una lista `list` y un elemento `elem`. El método `countApariciones` debe devolver el número de apariciones de `elem` en la lista `list` (y 0 en caso de que `elem` no esté en `list`). El parámetro `list` podrá ser **`null`** (en cuyo caso contiene cero apariciones de cualquier elemento) y podrá contener elementos **`null`**. Igualmente, el parámetro `elem` también podrá ser **`null`**. El uso de bucles (**`while`**, **`for`**, **`do-while`** o **`for`**`-each`) así como de iteradores NO está permitido. La implementación debe realizarse mediante un **método auxiliar** que sea **recursivo**. No está permitido modificar el contenido de `list`.

Por ejemplo, dado `list = [1,2,3,null,2,4,5,null]`, la llamada `countApariciones(list,18)` debe devolver 0; `countApariciones(list,2)` debe devolver 2; `countApariciones(list,null)` debe devolver 2.

Ejercicio 49: Método `multiplyAndClean`

fácil

Implementar en Java el método:

```
static <E> void multiplyAndClean (PositionList<Integer> list, int n)
```

que recibe como parámetro una lista `list` y un número `n`. `list` nunca será **`null`**, pero puede contener elementos **`null`**. El método debe multiplicar por `n` todos los elementos de la lista que no sean **`null`** y eliminar los que si lo sean.

Por ejemplo, si `list1 = [1,2,2,null,1,null]`, la llamada `multiplyAndClean(list1,3)` debe dejar `list1 = [3,6,6,3]`. Si `list2 = [null,2,2,null,null,null]`, la llamada `multiplyAndClean(list2,2)` debe dejar `list2 = [4,4]`.

Ejercicio 50: Completando `copiarHastaElem` recursivo

fácil

El método `copiarHastaElem` recibe como parámetro una lista `list` y un elemento `elem` y debe devolver una **nueva** lista con los elementos de `list`, en el mismo orden, hasta la primera aparición de `elem`, pero sin incluirlo. Si `elem` no está en `list`, `copiarHastaElem` debe devolver una copia de `list`. Si `list` es **`null`**, se deberá lanzar la excepción `IllegalArgumentException`. Por otro lado, `list` no contendrá elementos **`null`** y `elem` no podrá ser **`null`**.

Por ejemplo, dado `list = [1,3,2,4,6]`, la llamada `copiarHastaElem(list,4)` devolverá `[1,3,2]`, `copiarHastaElem(list,9)` devolverá `[1,3,2,4,6]`.

Completar todos los huecos marcados con `/**** HUECO XX ****/` en la siguiente implementación recursiva de `copiarHastaElem`:

```
static <E> PositionList<E> copiarHastaElem(PositionList<E> list, E elem) {
  PositionList<E> res = new NodePositionList<E>();

  /**** HUECO 1 ****/
```

```
  copiarHastaElemRec(list,elem,/**** HUECO 2 ****/);
  return res;

}

static <E> void copiarHastaElemRec(PositionList<E> list,
                                   E elem,
                                   Position<E> cursor,
                                   PositionList<E> res) {
  if (/**** HUECO 3 ****/) {
    return;
  }

  res.addLast(cursor.element());
  copiarHastaElemRec(/**** HUECO 4 ****/);
}
```

Ejercicio 185: Método `copiarHastaSumN`

fácil

Implementar en Java el método:

```
PositionList<Integer> copiarHastaSumN (PositionList<Integer> list, int n)
```

que recibe como parámetro una lista `list`, cuyos elementos serán siempre enteros positivos y estarán ordenados de forma creciente, y un número `n`. El método debe devolver una **nueva** lista que contenga el mayor número posible de elementos contenidos en `list`, en el mismo orden en que aparecen en `list`, y cuya suma no supere `n`. Cuando `n` sea un número ≤ 0 el método deberá lanzar la excepción `IllegalArgumentException`. La lista `list` no contendrá elementos **`null`**. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, dado `list = [2,3,3,4,5,6]`, la llamada `copiarHastaSumN(list,8)` devolverá `[2,3,3]`, `copyElemsNbyN(list,12)` devolverá `[2,3,3,4]`, `copyElemsNbyN(list,65)` devolverá `[2,3,3,4,5,6]` y `copyElemsNbyN(list,1)` devolverá `[]`.

Ejercicio 52: Método `sumaElementosPositivos` (recursivo)

fácil

Implementar de forma **recursiva** en Java el método:

```
static int sumaElementosPositivos (PositionList<Integer> list)
```

que recibe como parámetro una `PositionList` y debe devolver **la suma de los elementos positivos** contenidos en la lista `list`. El uso de bucles (**`while`**, **`for`**, **`do-while`** o **`for`**`-each`), así como de iteradores, o la creación de estructuras de datos nuevas como pilas, colas, listas, Strings, etc, NO está permitido. La implementación debe realizarse mediante un **método auxiliar** que sea **recursivo**. El argumento `list` podrá ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. Además `list` podrá contener elementos elementos **`null`**.

Por ejemplo, si se invoca `sumaElementosPositivos(list)` sobre la lista `list = [0,1,-2,3,null,3]`, el método deberá devolver `7`; si `list = [5,-2]`, deberá devolver `5`; y si `list = [null,null]` deberá devolver `0`.

Ejercicio 53: Método `copiaCircular` fácil

Implementar en Java el método:

```
static <E> PositionList<E> copiaCircular (PositionList<E> list, Position<E> pos)
```

que recibe como parámetro una lista `list` y una posición de la lista `pos`. El método debe crear una **nueva** lista que contenga los mismos elementos que `list`, en la que el primer elemento de la lista resultado será el elemento contenido en `pos` y los siguientes elementos de la lista resultado se obtendrán recorriendo el resto de la lista `list` de forma circular, empezando en el siguiente a `pos` y cuando se alcance el final debe regresar al principio hasta llegar de nuevo a la posición `pos` (sin incluir su elemento de nuevo). La lista `list` podrá ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, dado `list = [2,3,3,4,5,6]`, la llamada `copiaCircular(list,pos(4))` debe devolver una lista con `[4,5,6,2,3,3]`, `copiaCircular(list,pos(2))` debe devolver `[2,3,3,4,5,6]` y `copiaCircular(list,pos(6))` debe devolver `[6,2,3,3,4,5]`.

Ejercicio 54: Método `multiplyAndClean` (recursivo) fácil

Implementar de forma **recursiva** en Java el método:

```
static <E> PositionList<Integer> multiplyAndClean (
  PositionList<Integer> list, int n)
```

que recibe como parámetro una lista `list` (que puede contener elementos **`null`**) y un entero `n` y debe devolver una **nueva** lista con los elementos de `list` que sean distintos de **`null`** multiplicados por `n` y manteniendo el orden que tienen los elementos en `list`. El uso de bucles (**`while`**, **`for`**, **`do-while`** o **`for-each`**) **NO está permitido**. La implementación debe realizarse mediante un **método auxiliar** que sea **recursivo**. El argumento `list` podrá ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, si se invoca `multiplyAndClean(list,2)` donde `list = [0,1,-2,3,null,3]`, el método deberá devolver una lista con los elementos `[0,2,-4,6,6]`.

Ejercicio 55: Método `sumar` fácil

Implementar en Java el método:

```
PositionList<Integer> sumar (PositionList<Integer> l1, PositionList<Integer> l2)
```

que recibe como parámetro dos listas `l1` y `l2` que representan números y debe devolver una **nueva** lista que represente la suma de `l1` y `l2`. Las listas `l1` y `l2` representan números positivos en base decimal y todos sus elementos serán números en el rango 0 a 9. Por ejemplo, la lista `[1,2,4]` representa el número 124 y la lista `[5,2,1]` representa el número 521. Ninguna de las listas de entrada será **`null`** ni contendrá elementos **`null`**. Ninguna lista de entrada estará vacía ni contendrá dígitos no significativos: la representación `[0,1]` no es válida (pero `[0]` sí). El resultado tampoco debe contener dígitos no significativos. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, `sumar([1,2],[7])` deberá devolver `[1,9]`, `sumar([1,2],[2,3])` deberá devolver `[3,5]`, `sumar([1,2],[1,9])` deberá devolver `[3,1]`, `sumar([9,9],[9,9])` deberá devolver `[1,9,8]` y `sumar([7,8],[1,2,3])` deberá devolver `[2,0,1]`.

Ejercicio 56: Método `cuantosEnRango` (recursivo)

fácil

Implementar de forma **recursiva** en Java el método:

```
static int cuantosEnRango (PositionList<Integer> list, int a, int b)
```

que recibe como parámetro una `PositionList` y un rango definido por `a` y `b`, y debe devolver el número elementos de `list` cuyo valor se encuentra dentro del rango `[a,b]`. Un valor `x` está dentro del rango si $a \leq x \leq b$. El argumento `list` nunca será **`null`** ni contendrá elementos **`null`**.

El uso de bucles (**`while`**, **`for`**, **`do-while`** o **`for`**-each), así como de iteradores, o la creación de estructuras de datos nuevas como pilas, colas, listas, Strings, etc, NO está permitido. La implementación debe realizarse mediante un **método auxiliar** que sea **recursivo**.

Por ejemplo, dada la lista `list = [5,4,2,6,7,2,1,3]`, si se invoca `cuantosEnRango(list,2,4)` el método deberá devolver `4`; si se invoca `cuantosEnRango(list,1,2)` deberá devolver `3`; si se invoca `cuantosEnRango(list,3,3)` deberá devolver `1`; o si se invoca `cuantosEnRango(list,4,2)` deberá devolver 0.

Ejercicio 57: Método `sumaListas`

fácil

Implementar en Java el método:

```
PositionList<Integer> sumaListas (PositionList<PositionList<Integer>> lista)
```

que recibe como parámetro `lista`, una lista de listas que contienen enteros, y debe devolver una **nueva** lista que contenga, para cada lista contenida en `lista`, la suma de su contenido. Por tanto, dada una lista $[l_0, l_1, ..., l_n]$ donde cada l_i es, a su vez, una lista de enteros, el método debe devolver $[sum(l_0), sum(l_1), ..., sum(l_n)]$ donde $sum(l)$ es la suma de los elementos contenidos en l. El argumento `lista` nunca será **`null`**. Ninguna de las listas de entrada l_i será **`null`** ni contendrá elementos **`null`**. En caso de que `lista` sea vacía, `sumaListas` debe devolver la lista vacía. En caso de que una de las listas l_i esté vacía, debe devolverse un 0 en la posición correspondiente.

Por ejemplo, la llamada `suma([[1,2,4],[4,5],[7,3,2,1]])` método debe devolver `[7,9,13]`, `suma([[-1,4],[-5],[6,4],[]])` debe devolver `[3,-5,10,0]` o `suma([])` debe devolver `[]`.

Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía. Recordad que el uso de métodos auxiliares está permitido.

Ejercicio 58: Método `hayEnRango` (recursivo)

fácil

Implementar de forma **recursiva** en Java el método:

```
static boolean hayEnRango (PositionList<Integer> list, int a, int b)
```

que recibe como parámetro una `PositionList` y un rango definido por `a` y `b`, y debe devolver **`true`** si existe un elemento en `list` que se encuentre dentro del rango `[a,b]`. Un valor `x` está dentro del rango si $a \leq x \leq b$. En caso de que el rango no sea válido, es decir, en caso de que $b < a$, el método simplemente debe devolver **`false`**, no debe lanzar ninguna excepción. El argumento `list` nunca será **`null`** ni contendrá elementos **`null`**.

El uso de bucles (**`while`**, **`for`**, **`do-while`** o **`for`**-each), así como de iteradores, o la creación de estructuras de datos nuevas como pilas, colas, listas, Strings, etc, NO está permitido. La implementación debe realizarse mediante un **método auxiliar** que sea **recursivo**.

Por ejemplo, dada la lista `list = [5,4,2,6,7,2,1,3]`, la llamada `hayEnRango(list,2,4)` debe devolver **`true`**; `hayEnRango(list,8,10)` debe devolver **`false`** o `hayEnRango(list,7,10)` debe devolver **`true`**.

Ejercicio 59: Método `elementosUnicos`

para pensar

Implementar en Java el método:

```
static <E> PositionList<E> elementosUnicos (PositionList<E> lista)
```

que recibe como parámetro una lista `lista` y debe devolver una **nueva** lista que contenga, en el orden de aparición en `lista`, los elementos que únicamente aparecen una vez en `lista`. La lista de entrada nunca será `null`, pero podrá contener elementos `null` que deben ser tratados como un elemento más de la lista. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, `elementosUnicos([1,2,7])` deberá devolver `[1,2,7]`,
`elementosUnicos([1,2,2,3])` deberá devolver `[1,3]`,
`elementosUnicos([])` deberá devolver `[]`,
`elementosUnicos([9,9,9])` deberá devolver `[]` y
`elementosUnicos([7,8,null,7])` deberá devolver `[8,null]`.

Ejercicio 60: Método `barajar` (recursivo)

medio

Implementar de forma **recursiva** en Java el método:

```
static <E> PositionList<E> barajar (PositionList<E> list)
```

que recibe como parámetro una lista `list` y devuelve una **nueva** lista que contiene los elementos contenidos en `list` en el siguiente orden: primero, último, segundo, penúltimo, tercero, antepenúltimo,... y así hasta completar todos los elementos de la lista. La lista `list` nunca será `null`, pero contendrá al menos un elemento y su tamaño debe ser un número impar (`l.size()%2 == 1`). En caso de recibir una lista con un número par de elementos, el método debe lanzar la excepción `IllegalArgumentException`. Por ejemplo, dado `list = [1,2,3,8,4,5,6]`, la llamada `barajar(list)` devolverá `[1,6,2,5,3,4,8]`, si `list = [1,2,null,7,4,5,null]` devolverá `[1,null,2,5,null,4,7]`. Se dispone de la clase `NodePositionList` que implementa el interfaz `PositionList<E>` y que tiene un constructor sin parámetros.

El uso de bucles (`while`, `for`, `do-while` o `for`-each), así como de iteradores, o la creación de estructuras de datos nuevas como pilas, colas, listas, Strings, etc, NO está permitido. La implementación debe realizarse mediante un **método auxiliar** que sea **recursivo**.

Ejercicio 61: Método `elementosRepetidos`

para pensar

Implementar en Java el método:

```
public static <E> PositionList<E> elementosRepetidos (PositionList<E> lista)
```

que recibe como parámetro una lista `lista` y debe devolver una **nueva** lista que contenga, en el orden de aparición en `lista` y sin que se repitan, los elementos que aparezcen más de una vez en ella. La lista de entrada nunca será `null`, pero podrá contener elementos `null` que deben ser tratados como un elemento más de la lista. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía. Para implementarlo podéis asumir que disponéis (sin necesidad de implementarlo) del método `member(PositionList<E> l, E e)`, que devuelve `true` si el `e` está en la lista `l`

Por ejemplo, `elementosRepetidos([1,2,7])` deberá devolver `[]`,
`elementosRepetidos([1,2,2,3])` deberá devolver `[2]`,
`elementosRepetidos([])` deberá devolver `[]`,

`elementosRepetidos([9,9,9])` deberá devolver `[9]` y
`elementosRepetidos([7,8,null,7,null])` deberá devolver `[7,null]`.

Ejercicio 62: Método `intercalar` (recursivo)

fácil

Implementar de forma **recursiva** en Java el método:

```
public static <E> PositionList<E> intercalar (PositionList<E> l1,
                                              PositionList<E> l2)
```

que recibe como parámetro dos listas de tipo `PositionList<Integer>`. El método debe devolver una *nueva* lista que contenga los elementos de las listas `l1` y `l2` intercalándolos de forma alterna y empezando por los elementos de `l1`. Si `l1` y `l2` tienen distinto tamaño, el método debe lanzar la excepción `IllegalArgumentException`. La longitud de la lista resultado debe ser la misma que la suma de las longitudes `l1` y de `l2`. Se asume que `l1` y `l2` no son null, pero pueden estar vacías.

Por ejemplo, si `l1` es la lista formada por los elementos [1,2,3,4] y `l2` por los elementos [5,6,7,8], el método `intercalar` devolverá una nueva lista con los elementos [1,5,2,6,3,7,4,8]. Si `l1` es [6,1] y `l2` es [3,4,8,7], el método debe lanzar la excepción `IllegalArgumentException`.

El uso de bucles (**`while`**, **`for`**, **`do-while`** o **`for`**-each), así como de iteradores, o la creación de estructuras de datos nuevas como pilas, colas, listas, Strings, etc, NO está permitido. La implementación debe realizarse mediante un **método auxiliar** que sea **recursivo**.

Ejercicio 63: Método `expandir`

fácil

Implementar en Java el método:

```
<E> PositionList<E> expandir (PositionList<Pair<E,Integer>> lista)
```

que recibe como parámetro una lista de pares de elementos `<E,Integer>` donde el primer elemento del par es un dato cualquiera y el segundo elemento es el número de veces consecutivas que el elemento aparece en esa posición de la lista. La lista podría contener dos veces el mismo elemento. El método debe devolver una **nueva** lista de elementos de tipo `E` que contenga, en el orden de aparición en `lista`, los elementos contenidos en `lista` repetidos tantas veces como indique el número de elemento contenido en el par.

La lista de entrada podrá ser **`null`**, en cuyo caso el método debe lanzar la excepción `IllegalArgumentException`. Asimismo, los pares podrán contener como elemento **`null`**, en cuyo caso deben ser tratados como cualquier otro elemento. El número de repeticiones de cada par nunca será **`null`** y será un número positivo.

El interfaz `Pair<A,B>` dispone de los métodos `getLeft()` y `getRight()`, que devuelven el primer y segundo elemento del par. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, `expandir([<a,3>,<b,2>,<a,4>])` deberá devolver una lista `[a,a,a,b,b,a,a,a,a]`, `expandir([<a,3>,<null,2>,<b,2>])` deberá devolver `[a,a,a,b,b]` o `expandir([])` deberá devolver una lista vacía `[]`.

Ejercicio 64: Método `expandirRec` (recursivo)

fácil

Implementar de forma **recursiva** en Java el método:

```
static <E> PositionList<E> expandirRec (PositionList<Pair<E,Integer>> lista)
```

que debe implementar la misma funcionalidad descrita en el Ejercicio anterior, pero la implementación debe hacerse de forma recursiva. El uso de bucles (**`while`**, **`for`**, **`do-while`** o **`for`**-each), así como de iteradores,

o la creación de estructuras de datos nuevas como pilas, colas, listas, Strings, etc, NO está permitido. Únicamente está permitida la creación de una lista de tipo `NodePositionList<E>` para devolver el resultado. La implementación debe realizarse mediante **métodos auxiliares** que sean **recursivos**.

Implementación de Listas

Ejercicio 65: Añadir elemento a lista con nodos enlazados

fácil

El siguiente método debería añadir un elemento `elem` a una lista construida con nodos enlazados colocándolo **antes** del nodo apuntado por el parámetro `p`. Por simplicidad asumiremos que `p` es una referencia válida y no comprobamos si realmente apunta a un nodo dentro de nuestra lista. La lista está construida con centinelas al principio y al final de la misma, para simplificar el código. `numElts` es un atributo de la lista que guarda el número de elementos de la misma.

```
public void addBefore(Node<E> p, E elem) {
  numElts++;
  Node<E> newNode = new Node<E>(p.getPrev(), p, elem);
  p.getPrev().setNext(newNode);
  p.setNext(newNode);
}
```

Sin embargo, este código tiene un error. ¿Cuál es y cómo lo corregirías?
Nota: no es un error de tipos o de sintaxis; es un error que hace que funcione mal, no que no compile.

Ejercicio 66: Insertar antes de una posicíon

fácil

Escribir el código Java del siguiente método de la clase `NodePositionList<E>` que implementa el interfaz `PositionList<E>`:

```
public void addBefore(Position<E> p, E e) throws InvalidPositionException;
```

Dicho método debe insertar un elemento `e` antes de la posición `p` en la lista implementada mediante nodos enlazados (interfaz `Node<E>`). Por simplicidad, se podrá asumir que `p` es una posición válida pudiéndose omitir el código que realiza dicha comprobación y lanza la excepción.

Ejercicio 67: Preguntar sobre implementación de listas

fácil

Indique la afirmación correcta de entre las siguientes:

(a) La clase `NodePositionList<E>` utiliza nodos centinelas en los que se almacenan el primer y último elemento de la lista.

(b) La búsqueda (método `get`) para la estructura de datos `IndexedList<E>` tiene complejidad $O(1)$.

(c) El borrado (método `remove`) para la estructura de datos `ArrayIndexedList<E>` tiene complejidad $O(1)$.

(d) El borrado (método `remove`) de un elemento en una `NodePositionList<E>` tiene complejidad $O(1)$.

Ejercicio 68: Añadir append a `NodePositionList` medio

Se decide añadir a la clase `NodePositionList<E>` el método `append`:

```
public class NodePositionList<E> implements PositionList<E> {
  protected int numElts;              // Number of elements in the list
  protected Node<E> header, trailer;  // Special sentinels
  ...
  void append(NodePositionList<E> list) { /** COMPLETAR **/ }
}
```

Implementar en Java el método `append` que será similar al implementado en clase de laboratorio con la salvedad de que tras concatenar los nodos `append` debe dejar la lista parámetro `list` vacía. Recordamos que `append` debe concatenar los nodos de `list` a los nodos de la lista sobre la que se invoca el método. La concatenación debe hacerse con complejidad constante, sin copiar los elementos de `list`.

Por ejemplo, se tienen las listas `l1` y `l2` y se invoca `l1.append(l2)`. Si `l2` es vacía entonces el método no tiene efecto, quedando `l1` inalterada. Si `l1` es vacía entonces `l1` debe «quedarse» con los nodos de `l2` y esta última debe quedar vacía. Si las dos listas contienen elementos, el siguiente nodo al último nodo de `l1` debe ahora ser el primer nodo de `l2`, es decir, los nodos de `l2` se concatenan a los de `l1`. La lista `l2` debe quedar vacía.

Ejercicio 69: Implementar el método `swap` fácil

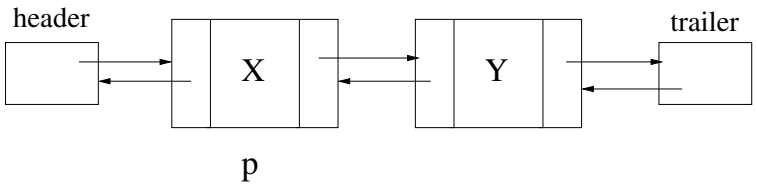
Se desea añadir a la clase `NodePositionList<E>` el siguiente método:

```
void swap(NodePositionList<E> list)
```

el cual debe intercambiar los elementos entre la lista parámetro y la lista sobre la que se invoca el método, pero debe hacerlo con complejidad constante en tiempo. Por ejemplo, sean `l1` y `l2` dos variables que referencian dos objetos lista distintos. Supongamos que `l1` referencia una lista con elementos 3,4,5 y `l2` una lista con elementos 0,7. Tras la invocación `l1.swap(l2)`, `l1` debe referenciar una lista con elementos 0,7 y `l2` una lista con elementos 3,4,5. El intercambio de elementos debe realizarse con complejidad constante. La invocación de swap sobre la misma lista, p.ej., `l1.swap(l1)`, debe dejar la lista intacta.

Ejercicio 70: Borrar siguiente fácil

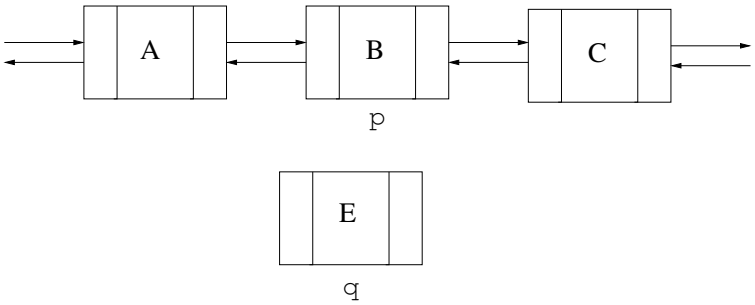
La siguiente figura muestra una lista de nodos `NodePositionList`. Los objetos nodo son de clase `Node<E>`:



La variable `p` referencia el nodo con el elemento `X`. Escribir las líneas de código que borran el nodo inmediatamente siguiente al nodo `p`. La solución no debe hacer uso de las variables header y trailer.

Ejercicio 71: Insertar después de un nodo fácil

La siguiente figura muestra un fragmento de una lista doblemente enlazada de objetos de clase `Node<E>`. Asumimos que las variables `p` y `q` almacenan los objetos nodo indicados.



Escribir las líneas de código que insertan el nodo `q` inmediatamente después del nodo `p` en la cadena doblemente enlazada, de forma que la lista contenga la secuencia de elementos A, B, E, C.

Ejercicio 72: Complejidad de inserción en `IndexedList`

fácil

Dada la implementación de listas indexadas `IndexedList` mediante vectores `ArrayIndexedList`. Indicar de entre las siguientes posibilidades la que describe correctamente la complejidad del método de inserción `add` para el caso peor. Se asume que n es el tamaño de la lista:

(a) $O(1)$ porque añadir un elemento a un vector de Java siempre tiene complejidad constante.

(b) $O(n)$ porque en el caso peor hay que desplazar todos los elementos del vector a la izquierda para hacer hueco al nuevo elemento.

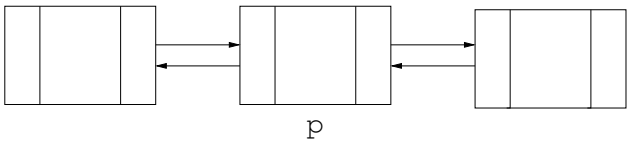
(c) $O(n)$ porque en el caso peor hay que desplazar todos los elementos del vector a la derecha para hacer hueco al nuevo elemento.

(d) $O(n)$ porque en el caso peor siempre hay que copiar los elementos del vector a un nuevo vector más grande y luego desplazar los elementos.

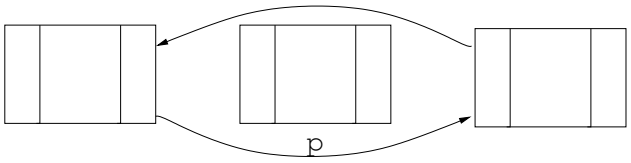
Ejercicio 73: Desconectar un nodo

fácil

Sea `p` una variable de tipo `Node<E>` que referencia un nodo de una lista tal y como se indica en el siguiente diagrama.



Escribir las líneas de código que desconectan el nodo `p` de la lista, tal y como sugiere el siguiente diagrama.



Ejercicio 74: Errores en `addFirst`

fácil

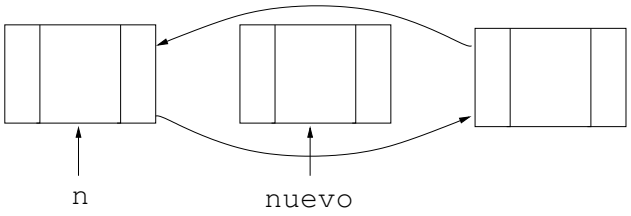
Indicar los dos errores de esta implementación incorrecta del método `addFirst`.

```
public void addFirst(E element) {
  Node<E> n = new Node<E>(header, trailer, element);
  header.getNext().setPrev(n);
  header.setNext(n);
}
```

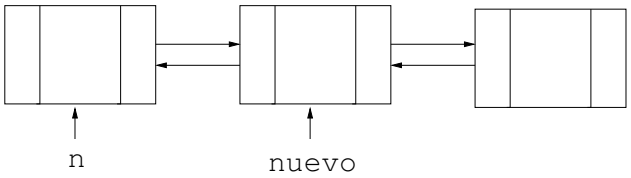
Ejercicio 75: Conectar un nodo

fácil

Se tiene una variable `n` de tipo `Node<E>` que referencia un objeto nodo de una lista de posiciones. Se tiene también una variable `nuevo` de tipo `Node<E>` que referencia un nodo que se desea insertar en la lista.



Escribir únicamente las líneas de código que conectan el nodo `nuevo` a la lista, tal y como indica el siguiente diagrama.



Pilas y Colas

Ejercicio 76: Invertir una cola

fácil

Implementar en Java el método

```
public static <E> FIFO<E> reverse (FIFO<E> q)
```

que toma como parámetro una cola FIFO `q` que no será null. Si la cola `q` es vacía entonces el método debe devolver una *nueva* cola `FIFO` vacía. En otro caso, el método `reverse` debe devolver una *nueva* cola en la que los elementos se encuentren en el orden inverso al orden de la cola `q` recibida como parámetro. Al terminar la ejecución del método, la cola `q` (si se ha modificado) debe tener los mismos elementos que tenía al empezar el método y éstos deben estar en el mismo orden. Para invertir el orden de los elementos de la cola es **obligatorio** utilizar una estructura temporal de tipo `LIFO<E>`.

Por ejemplo, sea `q` una cola `FIFO<E>` de elementos `Integer` de tamaño 4 que contiene los elementos [0,2,5,1] en la que el '0' sería el primer elemento en ser desencolado y el '1' el último elemento en ser desencolado. El método `reverse()` debe devolver una *nueva* cola FIFO con los mismos elementos, pero en orden inverso, es decir, [1,5,2,0], donde el '1' sería el primer elemento en ser desencolado y el '0' el último. Puede asumirse que se dispone de las clases `FIFOList<E>` y `LIFOList<E>` que implementan, respectivamente, los interfaces `FIFO<E>` y `LIFO<E>`.

Ejercicio 77: ¿Están los símbolos equilibrados?

para pensar

Implementar en Java el método

```
boolean estaEquilibrado (Character [] texto)
```

que toma como parámetro un «array» de caracteres `texto` que contendrá paréntesis y llaves de apertura: '(', '{' y de cierre ')', '}', así como otros caracteres que no serán **`null`**. El método devolverá **`true`** si los paréntesis y las llaves se encuentran correctamente equilibrados (balanceados) en `texto`. Una cadena con paréntesis está equilibrada si para cada paréntesis (llave) de apertura hay uno y solo un paréntesis (llave) de cierre del mismo tipo y cada aparición de un signo de cierre corresponde al último signo de apertura pendiente de cerrar. NOTA: La clase `Character` es la clase envoltorio correspondiente al tipo básico **`char`**.

Por ejemplo, dado el array ['a', '(', '}'] el método devolverá **`false`**, dado ['a', '(', '{', 'a', '}', 'b', 'c', ')'] devolverá **`true`**, dado ['a', '(', '{', ')', 'b', '}'] devolverá **`false`**, dado ['a', '(', '{', '}', 'b'] devolverá **`false`**, o dado ['a'] devolverá **`true`**.

Es obligatorio implementar el método utilizando una estructura `LIFO`. Se dispone de la clase `LIFOList` que implementa el interfaz `LIFO`. El array de entrada `texto` no será **`null`**.

Ejercicio 78: Implementando una pila

medio

Dada la definición de la clase `LIFOList<E>`, que implementa el interfaz `LIFO<E>` utilizando el atributo `elementos`, de tipo `PositionList<E>`, para guardar los elementos de dicha pila:

```
public class LIFOList<E> implements LIFO<E> {
  private PositionList<E> elementos;

  public LIFOList () {
    elementos = new NodePositionList<E>();
```

```
  }

  // Implementar los métodos push, pop, top, size y isEmpty!!
}
```

Codificar los métodos `push`, `pop`, `top`, `size` e `isEmpty` de la clase `LIFOList`. **IMPORTANTE: NO** debéis implementar los métodos `iterator` y `toArray`. **NO** está permitido crear atributos nuevos en la clase `LIFOList`. Recordad que la descripción completa del interfaz se encuenta en la hoja de interfaces.

Ejercicio 79: Implementando Atrás en un navegador

fácil

Se dispone de la clase `ControlNavegador` que permite gestionar el botón *atrás* de los navegadores de Internet (Google Chrome, Firefox, etc):

```
class ControlNavegador {
  private LIFO<String> pilaAtras = new LIFOList<String>();
  private String actual = null;

  public void nuevaURL (String url) {...}
  public String atras () throws AtrasNoPosibleException {...}
}
```

El atributo `pilaAtras` permite almacenar las URL's a las que se puede ir con el botón *atrás*. El atributo `actual` almacena la URL que está presentando el navegador.

Implementar en Java los métodos `nuevaURL` y `atras`. El método `nuevaURL` cambiará la URL actual y guardará la antigua URL entre las que se puede ir hacia atrás. El método `atras` devolverá la URL inmediatamente anterior a la actual, cambiará la URL actual por dicha URL inmediatamente anterior, descartando la URL que estaba como actual. En caso de que no haya direcciones para volver atrás, el método `atras` lanzará la excepción `AtrasNoPosibleException`. Nótese que, al crear el objeto, la URL `actual` es **`null`** y ésta no debe ser guardada para volver atrás ya que no es una URL válida.

Ejercicio 80: Implementando una cola

medio

Dada la definición de la clase `FIFOList<E>`, que implementa el interfaz `FIFO<E>` utilizando el atributo `elementos`, de tipo `PositionList<E>`, para guardar los elementos contenidos en la cola:

```
public class FIFOList<E> implements FIFO<E> {
  private PositionList<E> elementos;

  public FIFOList () {
    elementos = new NodePositionList<E>();
  }

  // Implementar los métodos enqueue, dequeue, first, size y isEmpty!!
}
```

Codificar los métodos `enqueue`, `dequeue`, `first`, `size` e `isEmpty` de la clase `FIFOList`. **IMPORTANTE: NO** debéis implementar los métodos `iterator` y `toArray`. **NO** está permitido crear atributos nuevos en la clase `FIFOList`. Recordad que la descripción completa del interfaz se encuenta en la hoja de interfaces.

Iteradores

Ejercicio 81: Iterador Positivos

medio

Dada la siguiente implementación del iterador `PositivePositionListIterator`.

```
public class PositivePositionListIterator<E extends Integer>
                                  implements Iterator<Integer> {

  private PositionList<Integer> list;
  private Position<Integer> cursor;

  public PositivePositionListIterator(PositionList<Integer> list) {
    if (list == null) {
      throw new IllegalArgumentException ("list no puede ser null");
    }
    this.list   = list;
    this.cursor = list.first();
    moveCursor(); /*AQUI*/
  }

  public boolean hasNext() {
    return cursor != null;
  }

  public Integer next() throws NoSuchElementException {
    if (cursor == null) {
      throw new NoSuchElementException();
    }

    Integer e = cursor.element();
    cursor = list.next(cursor);
    moveCursor(); /*AQUI*/
    return e;
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }

  private void moveCursor() {
     /* IMPLEMENTAR */
  }
}
```

Dicho iterador permite iterar sobre una lista de tipo `PositionList<Integer>` devolviendo únicamente aquellos elementos de la lista que sean mayores que 0. La lista a iterar podrá contener elementos **`null`**. Por ejemplo, dada la lista [-1,0,1,3,-2,7,null] el iterador devolverá los elementos 1,3,7 en este orden, o dada la lista [null,-1,0] el iterador no devolverá ningún elemento. En la implementación dada, el cursor siempre debe estar

apuntando al siguiente elemento a devolver por el iterador o a **null** en caso de que no queden elementos por devolver, siendo el método `moveCursor` el encargado de darle el valor adecuado al atributo `cursor`.

Implementar el método **private void** `moveCursor()`, invocado en las líneas marcadas con `/*AQUI*/`, que avanza el atributo `cursor` hasta el siguiente elemento a devolver por el iterador, o lo deja a **null** si no hay mas elementos que devolver.

Ejercicio 82: Iterador sobre no-nulos

fácil

Se desea implementar una clase `NonNullElementIterator<E>` de objetos iteradores sobre listas de posiciones `PositionList<E>`. Estos iteradores deben iterar únicamente sobre elementos de la lista distintos de **null**. Es decir, a diferencia del iterador ordinario de listas de posiciones que se ha visto en clase de teoría (clase `ElementIterator<E>`), el iterador `NonNullElementIterator<E>` debe «saltarse» los elementos **null** de la lista sobre la que itera. Por ejemplo, dado el siguiente método `show` que usa un iterador para mostrar por consola los elementos de una lista de enteros:

```
public static void show(PositionList<Integer> list) {
  Iterator<Integer> it = list.iterator();
  while(it.hasNext())
    System.out.print(it.next() + " ");
  System.out.print(".");
}
```

Si `list.iterator()` devuelve un objeto de clase `ElementIterator<Integer>` (el iterador ordinario visto en clase), la siguiente tabla muestra la salida por consola para cuatro listas dadas:

`show(lista1)`	salida por consola:	`1 6 7 8 .`
`show(lista2)`	salida por consola:	**null** `1 4` **null** `5` **null** `.`
`show(lista3)`	salida por consola:	**null null** `6 .`
`show(lista4)`	salida por consola:	**null null null** `.`

Sin embargo, si `list.iterator()` devuelve un objeto de clase `NonNullElementIterator<Integer>` entonces la salida por consola deberá ser diferente, pues el método `next()` de este iterador no debe devolver los elementos **null**, debe saltarlos:

`show(lista1)`	salida por consola:	`1 6 7 8 .`
`show(lista2)`	salida por consola:	`1 4 5 .`
`show(lista3)`	salida por consola:	`6 .`
`show(lista4)`	salida por consola:	`.`

A continuación se muestra el código incompleto de la clase `NonNullElementIterator<E>`. Dicha clase utiliza un iterador ordinario para recorrer los elementos de la lista, y un cursor al elemento en vez de al nodo. Completar el código del constructor y del método `next()`. Pueden definirse todos los métodos auxiliares que se considere necesario. No es necesario añadir más atributos a la clase. No es necesario, ni se puede, utilizar los métodos del interfaz `PositionList<E>`.

```
public class NonNullElementIterator<E> implements Iterator<E> {
  private ElementIterator<E> it;  /* utiliza un iterador ordinario */
  private E cursorElem;           /* cursor a un elemento */

  public NonNullElementIterator(PositionList<E> list) {
    if (list == null) throw new NullPointerException();
    else {
      this.it = list.iterator(); /* obtenemos el iterador ordinario */
      /** COMPLETAR **/
    }
  }
  public boolean hasNext() { return this.cursorElem != null; }
```

```
  public E next() throws NoSuchElementException {
    /** COMPLETAR **/
  }
}
```

Ejercicio 83: Iterador que salta un elemento

fácil

Se tiene la siguiente clase incompleta de objetos iteradores para listas de posiciones. Dichos iteradores deben iterar sobre una lista saltándose las apariciones de un elemento concreto que se pasa al constructor de la clase junto con la lista sobre la que iterar.

```
public class SkipIterator<E> implements Iterator<E> {
  private PositionList<E> list;  // lista sobre la que iterar
  private E elem;                // elemento a saltar
  private Position<E> cursor;    // cursor del iterador

  public SkipIterator(PositionList<E> list, E elem) {  // Constructor
    this.list = list;
    this.elem = elem;
    /** COMPLETAR **/
  }
  public boolean hasNext() { return this.cursor != null; }

  public E next() throws NoSuchElementException {
    if (this.cursor == null) throw new NoSuchElementException();
    else {
      E toReturn = this.cursor.element();
      /** COMPLETAR **/
      return toReturn;
    }
  }
}
```

El constructor toma una lista `list` que asumimos es distinta de null y no tiene elementos null, y toma un elemento `elem` que asumimos distinto de null. El constructor almacena la lista y el elemento en atributos de la clase. Después debe inicializar el cursor a la primera posición que contenga un elemento distinto de **`this`**`.elem`. Si no hay tal posición o se ha llegado al final de la lista entonces debe dejar el cursor a null.

El método `next()` devuelve el elemento en la posición a la que referencia el cursor, pero debe avanzar el cursor hacia la derecha hasta la primera posición que contenga un elemento distinto de **`this`**`.elem`. Si no hay tal posición o se ha llegado al final de la lista entonces debe dejar el cursor a null.

Completar el constructor y el método `next()` de la clase.

El siguiente fragmento de código ilustra un caso de uso del iterador en el que se pasa al constructor el entero 7 y una lista de enteros:

```
SkipIterator<Integer> it = new SkipIterator<Integer>(7,list);
while (it.hasNext()) System.out.println(it.next());
```

Si `list` tuviera los elementos 0, 4, 7, 8, 7, 7, 1 en ese orden entonces la salida por consola sería 0, 4, 8, 1. Si `list` tuviera los elementos 7, 7 en ese orden entonces no se mostraría nada.

Ejercicio 84: Método `minimo` con iteradores

fácil

Implementar en Java el método:

```
static Integer minimo (Iterable<Integer> iterable)
```

que recibe como parámetro un `Iterable` sobre elementos de tipo `Integer` y que devuelve el mínimo elemento del iterable. El iterable nunca será **`null`** ni devuelve un iterador **`null`**. El iterador devuelto por el `iterable` proporciona siempre elementos que son números naturales o el valor **`null`**, es decir, los elementos que no son **`null`** serán ≥ 0. En el caso de que `iterable` no contiene ningún elemento o sólo elementos **`null`**, el método debe devolver -1. **NO** está permitido el uso de **for-each** para recorrer los elementos de `iterable`.

Por ejemplo, dado un iterable que devuelve un iterador con los elementos `1,2,3,4,5,6`, la llamada al método devolverá `1`, si `iterable` devuelve un iterador con los elementos `1,null,0,8` el método devolverá `0`.

Ejercicio 85: Método `findPos` con iteradores

fácil

Implementar en Java el método

```
static <E> int findPos (Iterable<E> iterable, E elem)
```

que recibe un `Iterable` y un elemento `elem`, y devuelve la posición que ocupa `elem` en el iterable. Como se hace habitualmente, 0 corresponde a la primera posición. Si `elem` no está en el iterable, el método devolverá -1. El parámetro `iterable` no será **`null`** ni contendrá elementos **`null`** y el parámetro `elem` tampoco será **`null`**.

Por ejemplo, si `iterable` devuelve los elementos `6,4,3,7,4`, la llamada a `findPos(iterable,4)` devolverá 1, la llamada a `findPos(iterable,7)` devolverá 3 o la llamada a `findPos(iterable,9)` devolverá -1.

Ejercicio 86: Método `imprimirSinConsecutivosIguales` con iteradores

fácil

Implementar en Java el método:

```
static <E> void imprimirSinConsecutivosIguales (Iterable<E> iterable)
```

que recibe como parámetro un `Iterable` y que imprime los elementos contenidos en el `iterable`, pero imprimiendo únicamente una vez los elementos consecutivos iguales. El iterable nunca será **`null`** ni devolverá un iterador **`null`**. Los elementos devueltos por el `iterable` podrán ser **`null`** y se tratarán como los demás elementos.

Por ejemplo, dado un iterable que devuelve un iterador con los elementos `1,2,2,4,2,null,null,6`, la llamada al método imprimirá por consola los elementos `1,2,4,2,null,6`.

Ejercicio 87: Método `iguales` con iteradores

medio

Implementar en Java el método:

```
public static <E> boolean iguales (Iterable<E> iterable1,
                                   Iterable<E> iterable2)
```

que recibe dos objetos que implementan el interfaz `Iterable<E>` y que debe devolver **`true`** cuando todos los elementos contenidos en ambos iterables sean iguales, es decir, que los iterables devuelvan elementos iguales en las mismas posiciones y que ambos iteradores devuelvan el mismo número de elementos. Ninguno de los dos iterables será **`null`**, pero ambos podrán devolver elementos **`null`**.

Por ejemplo, dados los siguientes iterables que devuelven los elementos:
`it1: 1,2,null,4`
`it2: 1,2,null,4`
`it3: 1,2,null,4,5`
`it4: 1,2,null,4,null`

Los resultados de las diferentes llamadas a `iguales` devolverán:
`iguales(it1,it2) = true`
`iguales(it2,it3) = false`
`iguales(it2,it4) = false`
`iguales(it3,it4) = false`

Ejercicio 88: Método `areAllGreaterThan` con iteradores

medio

Implementar en Java el método

```
static <E> boolean areAllGreaterThan (Iterable<Integer> iter, Integer elem)
```

que recibe como argumentos `iter` (un `Iterable`) y un elemento `elem`, y devuelve **`true`** si todos los elementos devueltos por el iterable son estrictamente mayores que `elem`. Los parámetro `iter` y `elem` no serán **`null`**, pero `iter` podrá devolver elementos **`null`**, que deben ser ignorados. Si el iterable no devuelve ningún elemento, o todos los elementos que devuelve son **`null`**, el método debe devolver **`true`**.

Por ejemplo, si `iter` devuelve los elementos `6,4,3,7,4`, la llamada a `areAllGreaterThan(iter,2)` devolverá **`true`** y la llamada a `areAllGreaterThan(iter,5)` devolverá **`false`**.

Ejercicio 89: Método `primerDesordenado` con iteradores

medio

Implementar en Java el método:

```
public static Integer primerDesordenado (Iterable<Integer> iterable) {
```

que recibe un objeto que implementa el interfaz `Iterable<E>` y que debe devolver el primer elemento del iterador que no se encuentre ordenado de forma ascendente (suponiendo de que dos elementos consecutivos iguales se consideran ordenados). Se asume que el iterador nunca devolverá elementos **`null`**. En caso de que todos los elementos del iterador se encuentren ordenados o el iterador no devuelva ningún elemento, el método debe devolver **`null`**.

Por ejemplo, dados los siguientes iterables que devuelven los elementos:
`it1: 1,2,5,4`
`it2: 1,2,3,4`
`it3: 1,2,2,1`
`it4: 1,3,2,4`

Los resultados de las diferentes llamadas a `primerDesordenado` devolverán:
`primerDesordenado(it1) = 4`
`primerDesordenado(it2) = null`
`primerDesordenado(it3) = 1`
`primerDesordenado(it4) = 2`

Ejercicio 90: Implementar un `IteradorCircular`

medio

Implementar en Java la clase `IteradorCircular` cuya estructura es:

```
class IteradorCircular<E> implements Iterator<E> {
  private IndexedList<E> list;
  private int cursor;
  ... // NUEVO/S ATRIBUTOS

  public IteradorCircular(IndexedList<E> list, int initPos) { ... }
  public boolean hasNext() { ... }
  public E next() throws NoSuchElementException { ... }
}
```

Dicho iterador recibe en el constructor una lista de tipo `IndexedList<E>` y un número `initPos` que corresponde a la posición del primer elemento que debe ser devuelto por el iterador. El iterador devuelve todos los elementos contenidos en `list` recorriéndola circularmente: cuando alcance el final debe regresar al principio hasta llegar de nuevo a la posición `initPos`. En caso de que la `list` esté vacía o `initPos` esté fuera del rango de posiciones de `list`, el constructor debe lanzar la excepción `IllegalArgumentException`. Recordad que el método `next()` debe lanzar la excepción `NoSuchElementException` si no quedan más elementos por devolver. Nótese que es necesario añadir nuevos atributos a la clase `IteradorCircular` para implementar el problema propuesto (uno o más de uno, dependiendo de la solución elegida).

Por ejemplo, para `list = [2,3,3,4,5,6]` y los siguiente iteradores, se espera el comportamiento listado a continuación:
`it = new IteradorCircular(list,2)` devolverá `3 4 5 6 2 3`
`it2 = new IteradorCircular(list,0)` devolverá `2 3 3 4 5 6`
`it3 = new IteradorCircular(list,5)` devolverá `6 2 3 3 4 5`
`it3 = new IteradorCircular(list,83)` lanzará la excepción `IllegalArgumentException`.

Ejercicio 91: Método `maximo` de un iterable

fácil

Implementar en Java el método:

```
static Integer maximo (Iterable<Integer> iterable)
```

que recibe como parámetro un `Iterable` sobre elementos de tipo `Integer` y que retorna el máximo elemento devuelto por el iterable. El iterable nunca será **`null`**, ni devolverá un iterador **`null`**, ni devolverá elementos **`null`**. En el caso de que `iterable` no devuelva ningún elemento el método debe lanzar la excepción `IllegalArgumentException`.

Por ejemplo: dado un iterable que devuelve un iterador con los elementos `1,2,3,4,5,6`, una llamada a `maximo()` devolverá `6`. Si `iterable` devuelve un iterador con los elementos `-1,2,0,8`, una llamada a `maximo()` devolverá `8`.

Ejercicio 92: Método `recolectardeNenN` con iteradores

medio

implementar el método

```
static  <E> PositionList<E> recolectardeNenN (Iterable<E> iterable, int n)
```

que recibe como parámetro un `Iterable` y un número `n`. El método debe devolver una `PositionList` que incluye, en el mismo orden devuelto por el iterable, los elementos devueltos por el iterable que ocupan las posiciones múltiplo de `n`, es decir, las posiciones $0, n, 2*n, 3*n, ...$ Se asume que el primer elemento del `iterable` tiene la posición 0. Tanto el `Iterable`, como los elementos devueltos por el `Iterable` nunca serán **`null`**. En caso de que el valor de `n` no sea mayor que 0, el método debe lanzar la excepción `IllegalArgumentException`. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, dado un `iterable` que devuelve los elementos `8,2,5,2,3,3,4,5,6`, la llamada `recolectardeNenN(iterable,2)` devolverá una lista con `[8,5,3,4,6]`, `recolectardeNenN(iterable,3)` devolverá una lista con `[8,2,4]`, `recolectardeNenN(iterable,4)` devolverá una lista con `[8,3,6]`, `recolectardeNenN(iterable,-2)` lanzará la excepción `IllegalArgumentException`.

Ejercicio 93: Método `getDistancias` con iteradores

medio

Implementar en Java el método:

```
PositionList<Long> getDistancias (Iterable<Integer> iterable)
```

que recibe como parámetro una estructura de datos `iterable` y debe devolver una lista que contiene las *distancias* entre cada par de números consecutivos del iterable siguiendo el mismo orden de la lista. La distancia entre dos números $d(x, y)$ se calcula como la cantidad de unidades que hay entre x e y. Por ejemplo la distancia entre 5 y 1 es 4, la distancia entre -2 y -4 es 2, o la distancia entre -2 y 3 es 5. El tamaño de la lista resultado será siempre el número de elementos devueltos por el iterable menos 1. Dada una secuencia de elementos $[e_0, e_1, e_2, e_3, e_4]$ devueltos por el iterable, la lista resultante deberan contener $[d(e_0, e_1), d(e_1, e_2), d(e_2, e_3), d(e_3, e_4)]$. El `Iterable` de entrada nunca será **`null`** ni contendrá elementos **`null`**. Si el iterable de entrada tiene menos de 2 elementos, el método debe lanzar la excepción `IllegalArgumentException`.

Por ejemplo, dado un iterable que devuelve `[1,5,2,4]` el método debe devolver `[4,3,2]`, dado un iterable `[-1,-1,-6,-4]` el método debe devolver `[0,5,2]` o dado un iterable `[-5,2,1]` el método debe devolver `[7,1]` o dado `[2,2]` el método debe devolver `[0]`.

Ejercicio 94: Método `esSerieGeometrica` con iteradores

fácil

implementar el método

```
public static boolean esSerieGeometrica (Iterable<Integer> iterable)
```

que recibe como parámetro un `Iterable` que devuelve números enteros. El método debe devolver **`true`** si los elementos $e_0, e_1, e_2, e_3...$ forman una serie geométrica que cumple la siguiente propiedad: $e_1 = e_0 * 2$, $e_2 = e_1 * 2$, $e_3 = e_2 * 2$, es decir, que para un elemento cualquiera $e_n = e_{n-1} * 2$. Tanto el `Iterable`, como los elementos devueltos por el `Iterable` nunca serán **`null`**. Si el iterable devuelve menos de dos elementos, el método debe devolver **`true`**.

Por ejemplo, dado un `iterable` que devuelve los elementos `2,4,8,16,32`, la llamada `esSerieGeometrica(iterable)` debe devolver **`true`** y dado un `iterable` que devuelve los elementos `2,4,8,20,32`, la llamada `esSerieGeometrica(iterable)` debe devolver **`false`**.

Ejercicio 95: Método `getDistanciasMayores` con iteradores

fácil

implementar el método

```
PositionList<Pair<Integer,Integer>> getDistanciasMayores
                                    (Iterable<Integer> iterable, int min)
```

que recibe como parámetro un `Iterable` y un número `min`. El método debe devolver una `PositionList` de `Pair` que incluirá, en el mismo orden devuelto por el iterable, los pares de elementos consecutivos (x,y) cuya distancia sea mayor que el valor `min`. La distancia entre dos números $d(x,y)$ se calcula como la cantidad de unidades que hay entre x e y. Por ejemplo, la distancia entre 5 y 1 es 4, la distancia entre -2 y -4 es 2, y la distancia entre -2 y 3 es 5. Tanto el `Iterable`, como los elementos devueltos por el `Iterable` nunca serán **`null`**. Si el iterable devuelve menos de dos elementos, `getDistanciasMayores()` debe devolver una lista vacía. En caso de que el valor de `min` sea negativo, el método debe lanzar la excepción `IllegalArgumentException`. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía. Asimismo, se dispone del constructor `Pair<A,B>(a,b)` que permite crear un objeto de tipo `Pair` que contiene los elementos `a` y `b`.

Por ejemplo, dado un `iterable` que devuelve los elementos `8,-2,5,4,3,6`, la llamada `getDistanciasMayores(iterable,2)` devolverá una lista con `[<8,-2>,<-2,5>,<3,6>]`, `getDistanciasMayores(iterable,3)` devolverá una lista con `[<8,-2>,<-2,5>]`, `getDistanciasMayores(iterable,-2)` lanzará la excepción `IllegalArgumentException`.

Ejercicio 96: Método `compactar` con iteradores

fácil

implementar el método

```
static <E> PositionList<Pair<E,Integer>> compactar (Iterable<E> iterable)
```

que hace el proceso inverso al método anterior, es decir, recibe como parámetro un `Iterable` de elementos de tipo `E` y devuelve una `PositionList` de `Pair<E,Integer>` que incluirá, en el mismo orden devuelto por el iterable, los elementos devueltos por `iterable` y el número de apariciones consecutivas de los mismos. El `Iterable` nunca será **`null`**. Sin embargo, los elementos devueltos por el `Iterable` podrán ser **`null`** y deben ser tratados como cualquier otro elemento. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía. Asimismo, se dispone del constructor `Pair<A,B>(a,b)` que permite crear un objeto de tipo `Pair` que contiene los elementos `a` y `b`. Recuerda que el interfaz `Pair<A,B>` además de `getLeft()` y `getRight()`, también dispone de los métodos `setLeft(a)` y `setRight(b)` para actualizar el primer y segundo elemento del par.

Por ejemplo, dado un `iterable` que devuelve los elementos `a,a,a,b,b,c,c,c,d`, la llamada `compactar(iterable)` devolverá una lista con `[<a,3>,<b,2>,<c,3>,<d,1>]`, o dado un `iterable` que devuelve los elementos `a,a,a,null,null,a,a`, la llamada `compactar(iterable)` devolverá una lista con `[<a,3>,<null,2>,<a,2>]`.

Árboles

Ejercicio 97: Profundidad de todos los nodos

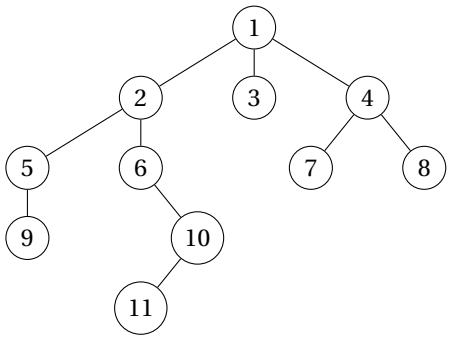
para pensar

Describe como utilizar alguno de los recorridos que conoces para calcular la profundidad de todos los nodos de un árbol general con complejidad $O(n)$, siendo n es el número de nodos.

Ejercicio 98: Recorridos en árbol general

fácil

Dado el siguiente árbol general donde los elementos de los nodos son números enteros:



Escribir la secuencia de elementos que corresponde a un recorrido en preorden y a un recorrido en postorden del árbol.

Ejercicio 99: Lista de nodos de un árbol cercanos a la frontera

medio

Implementar en Java el método

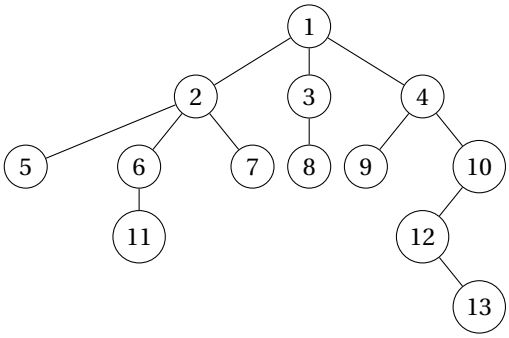
```
public PositionList<E> getExternalChildren(Tree<E> t)
```

que toma como argumento un árbol general y devuelve un objeto `NodePositionList` cuyos elementos son todos los nodos del árbol que tienen al menos un hijo nodo externo.

Ejercicio 100: Elementos en preorden y postorden

fácil

Dado el siguiente árbol general con números almacenados en sus nodos:



Escribir la secuencia de elementos que corresponde a un recorrido en preorden y a un recorrido en postorden del árbol. Nótese que dicha secuencia debe ser idéntica a la mostrada por la salida de los métodos `toStringPreorder(t,t.` y `toStringPostorder(t,t.root())`, donde `t` es el árbol.

Ejercicio 101: Lista de nodos externos de un árbol

fácil

Implementar el método

```
public <E> PositionList<E> leaves (Tree<E> t)
```

que toma como argumento un árbol general y devuelve un `PositionList<E>` con todos los elementos de los nodos externos (hojas) del árbol.

Ejercicio 102: Nodos hermanos

fácil

Implementar en Java el método:

```
public boolean siblings(Tree<E> t, Position<E> p1, Position<E> p2)
```

que toma como parámetro un árbol `t` y dos nodos `p1` y `p2` de dicho árbol. El método debe indicar si dichos nodos son hermanos.

Ejercicio 103: Ancestros de un nodo en un árbol

fácil

Implementar en Java el método

```
public static <E> PositionList<E> ancestros(Tree<E> t, Position<E> p)
```

que toma como parámetro un árbol no vacío `t` y un nodo `p` de dicho árbol. El método debe devolver como resultado una lista de posiciones con los elementos en los ancestros propios de `p`.

Ejercicio 104: Caminos hasta nodos hoja

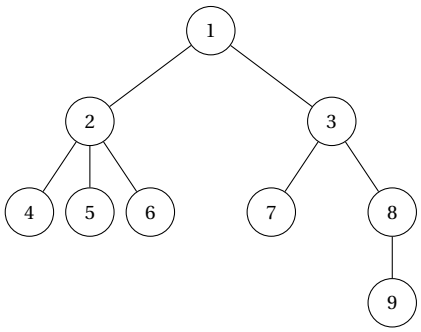
para pensar

Implementar en Java el método

```
static <E> void imprimirCaminosHojas(Tree<E> tree)
```

que recibe como parámetro un árbol `tree` e imprime todos los caminos del árbol que llevan desde la raiz hasta los nodos hoja (externos) del árbol, imprimiendo tanto la raiz como el nodo hoja. Los caminos que llevan desde la raiz a nodos internos no deben ser impresos. El árbol `tree` no será **`null`** y no contendrá elementos **`null`**. Para imprimir podéis usar `System.out.println(...)`.

Por ejemplo, dado el siguiente árbol `tree`, el método `imprimirCaminosHojas(tree)` debe imprimir:




```
> imprimirCaminosHojas(tree);

124
125
126
137
1389
```

Ejercicio 105: Cumple un árbol general la heap-order-property

medio

Implementar en Java un método con la cabecera

```
public static boolean hasHeapPropertyGen(Tree<Integer> tree)
```

Queremos implementar un método en Java que compruebe si un árbol general cumple la propiedad que caracteriza a los montículos (*heaps*):

> *El elemento contenido en cada nodo es mayor o igual que el elemento contenido en su padre, si tiene padre.*

Ejercicio 106: Método `maximoCamino`

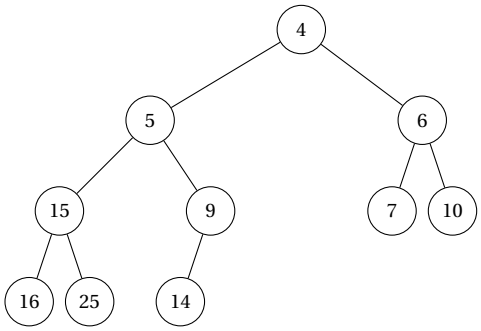
medio

En un árbol binario cuyos nodos contienen enteros no positivos, definimos el valor de un camino como la suma de los valores contenidos en los nodos que constituyen ese camino. Implementar en Java el método:

```
static Integer maximoCamino (BinaryTree<Integer> tree)
```

que recibe como parámetro el árbol binario `tree`, cuyos nodos contienen enteros **positivos**, y devuelve **el valor del camino que empieza en la raíz y cuyo valor es máximo**. Si `tree` es **`null`**, el método debe lanzar la excepción `IllegalArgumentException`. Los nodos de `tree` no contienen elementos **`null`**.

Por ejemplo, dado el árbol



el método debe devolver 49, ya que el camino con mayor valor desde la raíz es el que termina en la hoja con valor 25, y el método devuelve $4+5+15+25=49$. **Nota:** aunque pueden generarse caminos y elegir aquel con valor máximo, no es necesario hacerlo para resolver el problema.

Ejercicio 107: Método `member`

medio

Implementar **de forma recursiva** en Java el método:

```
static <E> boolean member (Tree<> tree, E elem)
```

que recibe como parámetro un árbol `tree` y un elemento `elem` y devuelve **`true`** si `elem` es un elemento del árbol `tree` y **`false`** en caso contrario. El elemento `elem` y el árbol `tree` nunca serán **`null`**, ni `tree` contendrá elementos **`null`**.

NOTA: **NO está permitido** usar **el método `iterator()`** para recorrer los elementos del árbol `tree`.

Ejercicio 108: Método `existeHoja`

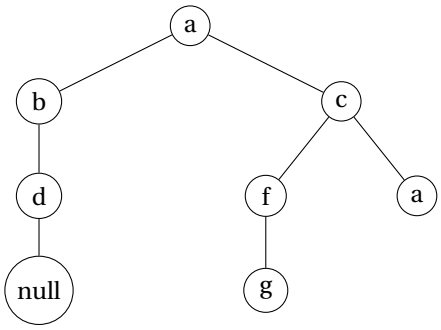
medio

implementar el método

```
public static boolean existeHoja (Tree<E> tree, E e)
```

que recibe como parámetro un árbol `Tree<E>` y un elemento `e` y debe devolver **`true`** si existe un nodo hoja igual que `e` El árbol `tree` nunca será **`null`**, pero podrá contener elementos **`null`**. Asimismo `e` podrá ser **`null`** y debe tratarse como cualquier otro elemento.

Por ejemplo, dado el siguiente árbol `tree`:



la llamada `existeHoja(tree,d)` devolverá **`false`**, la llamada `existeHoja(tree,g)` devolverá **`true`**, la llamada `existeHoja(tree,null)` devolverá **`true`**, y la llamada `existeHoja(tree,a)` devolverá **`true`**.

Árboles binarios

Ejercicio 109: Hoja más a la izquierda

medio

Dado un árbol binario, escribir un método que devuelva el valor almacenado en el nodo hoja situado más a la izquierda en el árbol. El método debe lanzar una excepción si el árbol está vacío. No podeis utilizar el método `iterator()` del interfaz.

Ejercicio 110: Comparación de árboles binarios

para pensar

Queremos comparar arboles binarios de elementos de tipo genérico `E` (interfaz `BinaryTree<E>`). Para ello definimos la clase `BinTreeComparator<E>` que implementa un comparador para dichos árboles.

`BinaryTree<E>` extiende el interfaz `Tree<E>` de la misma librería, y por tanto hereda el método `iterator()` que devuelve un iterador sobre los elementos del árbol.

El criterio para comparar dos `BinaryTree<E>` es:

1. Comparar elementos: recorrer ambos árboles simultáneamente mediante dos iteradores, comparando elemento a elemento con el comparador de elementos de tipo `E` que corresponda (este comparador de elementos es un atributo de la clase `BinTreeComparator<E>` que se inicializa en el constructor de esta clase). En el momento en el que dos elementos comparados sean distintos, la comparación de los árboles habrá terminado, y se considerará menor el árbol al cual pertenezca el menor de los citados dos elementos.
2. En caso de que todas las comparaciones de los elementos de los dos árboles, uno a uno y en el orden impuesto por el iterador, den la igualdad como resultado, el criterio para decidir qué árbol es mayor que otro o si son iguales, será comparar el número de elementos, es decir, si un árbol tiene más elementos que otro, se entenderá que es mayor.

(a) Sin necesidad de ver código, y siendo `n` el número de nodos del árbol que menos tenga, ¿crees que sería posible una implementación con complejidad $O(\log n)$ en el peor caso? Razona brevemente tu respuesta.

(b) Escribir el código del método compare de la clase `BinaryTreeComparator<E>` en el espacio destinado para ello a continuación:

```
public class BinaryTreeComparator<E> implements Comparator<BinaryTree<E>> {
   private Comparator<E> compElementos;

   public BinaryTreeComparator(Comparator<E> compElementos) {
        this.compElementos = compElementos;
   }

   // completar este método
   public int compare(BinaryTree<E> t1, BinaryTree<E> t2) {

    ...

   }
}
```

Ejercicio 111: Peor caso de un árbol binario de búsqueda

medio

De las siguientes secuencias de inserción en un árbol binario de búsqueda inicialmente vacío, indica la que genere el peor caso en base al número de niveles que queden en el árbol final:

(a) 1, 2, 3, 4, 6, 5, 7.

(b) 7, 1, 6, 2, 5, 3, 4.

(c) 4, 2, 6, 1, 5, 3, 7.

(d) 7, 6, 5, 4, 2, 1, 3.

Ejercicio 112: Averiguar recorrido en preorden de un árbol binario

fácil

Si al recorrer un árbol binario en inorden el resultado es `B A C K F H D E G`, marca cuál sería el recorrido en preorden:

(a) `FABKCDEHG`

(b) `BAFKHDCEG`

(c) `FBAKDCHEG`

(d) `FABKCDHGE`

Ejercicio 113: Áltura máxima y mínima de un árbol binario

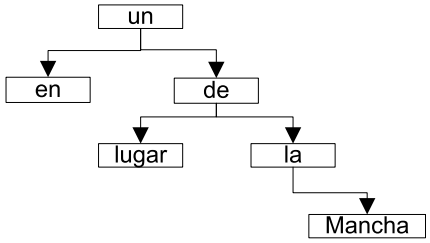
medio

Indicar la altura máxima y la altura mínima que puede tener un árbol binario del que solamente se sabe que tiene n nodos. Razone la respuesta.

Ejercicio 114: Código de creación de un árbol binario

fácil

Dado el árbol binario de la figura de la derecha, indique una secuencia de operaciones en Java que permita crearlo.



Ejercicio 115: Resultados de recorrido de un árbol binario

fácil

Dado el árbol del ejercicio 114 indique:

(a) el resultado de recorrer el árbol en *preorden*.

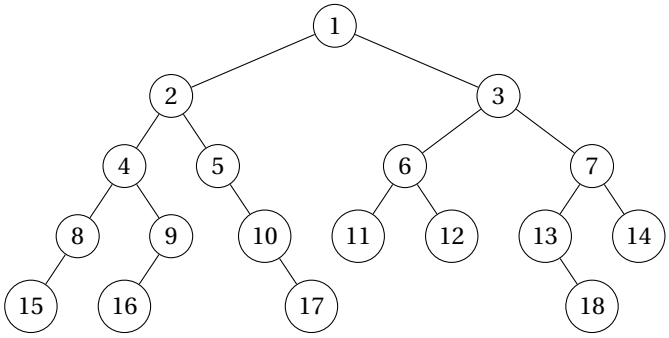
(b) el resultado de recorrer el árbol en *inorden*.

(c) el resultado de recorrer el árbol en *postorden*.

Ejercicio 116: Identificar recorridos de árbol

fácil

Dado el siguiente árbol binario en el que sólo se muestran los elementos en los nodos:



Indicar el tipo de recorrido (preorden, inorden, postorden, arbitrario) de los siguientes recorridos que muestran los elementos en los nodos:

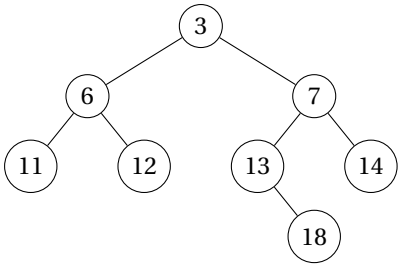
(a) 15 8 16 9 4 17 10 5 2 11 12 6 18 13 14 7 3 1

(b) 15 8 4 16 9 5 10 17 2 1 11 6 12 3 18 13 7 14

Ejercicio 117: Operaciones que crean un árbol binario

fácil

Indique una secuencia de operaciones en Java que permita crear el siguiente árbol binario:



Ejercicio 118: Código de impresión en inorden

fácil

Implementar en Java el método

```
public void inOrder(BinaryTree<E> t)
```

que toma como parámetro un árbol binario y muestra por pantalla la secuencia de elementos almacenados en los nodos según un recorrido en «inorden».

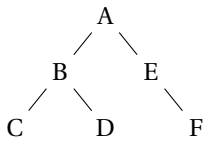
Ejercicio 119: Recorrido en anchura de un árbol

para pensar

Implementar en Java el método

```
public NodePositionList<E> breadth(BinaryTree<E> tree)
```

que devuelve una lista de posiciones con los elementos en los nodos del árbol según un recorrido en anchura (por niveles) del mismo. El recorrido en anchura de un árbol visita los nodos por niveles, primero el nivel 0 (la raíz), luego el nivel 1 (los hijos de la raíz de izquierda a derecha), luego el nivel 2, etc. Por ejemplo, la invocación de `breadth` sobre el árbol de la figura debe devolver la lista con los elementos A,B,E,C,D,F en ese orden de izquierda a derecha.



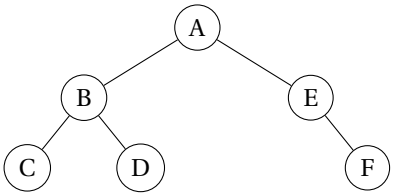
Ejercicio 120: Dejar sólo espina izquierda de un árbol binario

medio

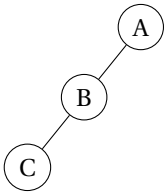
Se dispone de un método:

```
public void leftSpine(BinaryTree<E> tree)
```

que poda el árbol binario que toma como parámetro dejando únicamente su espina izquierda. La espina izquierda de un árbol binario t es el árbol binario degenerado que contiene únicamente los nodos de t en el camino de la raíz al nodo más a la izquierda. Por ejemplo, la espina izquierda del siguiente árbol:



es el siguiente árbol binario degenerado:



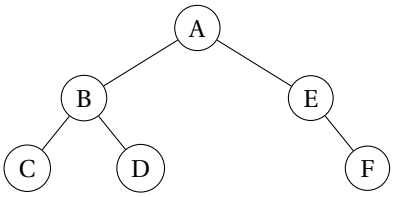
En cambio, si el árbol original no tuviera el nodo C, entonces la espina izquierda estaría formada por los nodos A y B.

1. Indicar la complejidad en caso peor que debería tener el método `leftSpine` suponiendo que el árbol binario está implementado mediante la clase `LinkedBinaryTree<E>`.
2. Implementar dicho método.

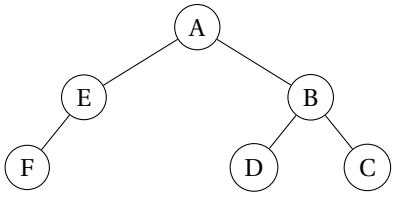
Ejercicio 121: Árbol especular

para pensar

Dado el siguiente árbol binario:



el árbol espejo de dicho árbol es el árbol obtiene visualmente dando la vuelta a esta hoja del examen, viéndose como a través de un espejo:



Implementar en Java el método

```
public <E> BinaryTree<E> espejo(BinaryTree<E> t)
```

que toma un árbol binario `t` devuelve su árbol espejo. como resultado.

Ejercicio 122: Identificar recorrido en postorden

fácil

Se tiene un árbol binario perfecto (todos los nodos internos tienen dos hijos) que almacena caracteres en los nodos. El recorrido del árbol en inorden produce la siguiente secuencia de elementos:

B D E A F G C

Indicar cuál de entre las siguientes secuencias de elementos es la que produce el recorrido en postorden de dicho árbol:

(a) A B C D E F G

(b) C E D F G E A

(c) B E D F C G A

(d) C G F A E D B

Ejercicio 123: Hermano izquierdo

fácil

Implementar en Java el método:

```
public Position<E> leftSibling(BinaryTree<E> t, Position<E> p)
  throws BoundaryViolationException
```

que toma como parámetro un árbol binario `t` y un nodo `p` de dicho árbol, y debe devolver el hermano izquierdo de `p`. Si `p` es un nodo que no tiene hermano izquierdo entonces el método debe lanzar la excepción.

Ejercicio 124: Hermando de nodo

fácil

Implementar en Java el método:

```
public Position<E> sibling(BinaryTree<E> tree, Position<E> p)
```

que toma como parámetro un árbol binario `tree` y un nodo `p` de dicho árbol, y debe devolver el hermano de `p`. Si `p` no tiene hermano entonces el método debe devolver **`null`**.

Ejercicio 125: Método misterioso

medio

Se ha implementado el siguiente método cuya funcionalidad se desconoce:

```
public Position<E> desconocido(BinaryTree<E> tree, Position<E> p)
{
  boolean done = false;
  while (!done) {
    if (tree.hasRight(p))
      p = tree.right(p);
    else if (tree.hasLeft(p))
      p = tree.left(p);
    else done = true;
  }
  return p;
}
```

El método `desconocido` recibe como parámetro un nodo `p` del árbol `tree`, y devuelve como resultado un nodo del árbol. Indicar la propiedad o característica del nodo devuelto por el método, sabiendo que el método siempre es invocado pasando en `p` la raíz del árbol `tree`.

Ejercicio 126: Hermano izquierdo

fácil

Implementar en Java el método:

```
public Position<E> leftSibling(BinaryTree<E> tree, Position<E> p)
```

que toma como parámetro un árbol binario `tree` y un nodo `p` de dicho árbol, y debe devolver el hermano izquierdo de `p`. Si `p` no tiene hermano izquierdo entonces el método debe devolver **`null`**.

Ejercicio 127: ¿Tiene hermano?

fácil

Implementar en Java el método

```
  public boolean hasSibling(BinaryTree<E> tree, Position<E> p)
```

que indica si un nodo `p` del árbol binario `tree` tiene un nodo hermano. Puede asumirse que `tree` y `p` no son **`null`** y que `p` es un nodo válido del árbol.

Ejercicio 128: Suma elementos de árbol

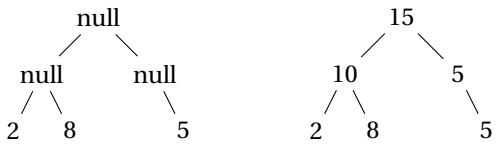
para pensar

Implementar en Java usando un método auxiliar recursivo el método:

```
  public static void sumTree(BinaryTree<Integer> tree)
```

que toma como parámetro un árbol binario `tree`. Si el árbol es null o vacío entonces el método no tiene efecto. En otro caso los elementos en los nodos hoja del árbol serán números enteros y los elementos en los nodos internos (en caso de haberlos) serán **`null`**. El método debe modificar el árbol para que cada nodo interno almacene como elemento el resultado de sumar el elemento en el nodo hijo izquierdo y el elemento en el nodo hijo derecho, después de que éstos hayan sido recursivamente tratados. Si un nodo interno tiene un único hijo entonces se almacena en el nodo interno el elemento del hijo después de ser tratado recursivamente.

Por ejemplo, el método `sumTree` debe modificar el árbol binario a la izquierda en la siguiente figura dejándolo como el mostrado a la derecha.



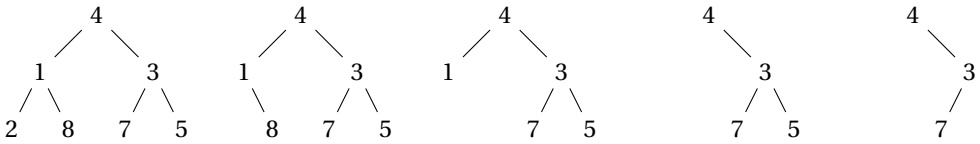
Ejercicio 129: Hoja más a la izquierda recursivamente

medio

Implementar usando un método auxiliar recursivo el método:

```
public static <E> Position<E> hojaMasIzquierda(BinaryTree<E> tree)
```

que devuelve el nodo hoja más a la izquierda del árbol binario `tree`. Si el árbol es **`null`** o vacío entonces el método devuelve **`null`**. El nodo hoja más a la izquierda del árbol es o bien el nodo raíz (si no tiene hijos) o bien es el nodo hoja más a la izquierda del subárbol que cuelga del nodo hijo izquierdo (si lo tiene) o bien el nodo hoja más a la izquierda del subárbol que cuelga del nodo hijo derecho (si no tiene nodo hijo izquierdo pero tiene nodo hijo derecho). Por ejemplo, en los siguientes árboles:



El nodo hoja más a la izquierda es respectivamente: 2, 8, 1, 7, 7.

Ejercicio 130: Contar recursivamente apariciones de un elemento en un árbol binario

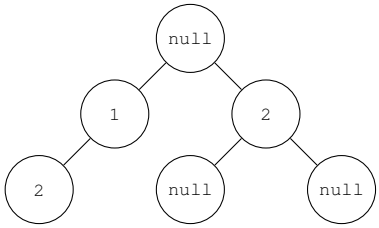
medio

Implementar **de forma recursiva** en Java el método

```
public static <E> int ocurrencias(BinaryTree<E> tree, E elem)
```

que toma como parámetro un árbol binario `tree` y un elemento `elem` e indica el número de ocurrencias del elemento `elem` en el árbol `tree`. El árbol `tree` no será **`null`** pero podrá contener elementos **`null`**. Si el valor que toma el parámetro `elem` es **`null`** se deberán contar el número de ocurrencias de **`null`** en el árbol. Es **obligatorio** implementar el método **de forma recursiva**.

A continuación se muestra un árbol `tree` y algunos de los resultados devueltos por el método `ocurrencias`:



`ocurrencias(tree,1)` devuelve 1
`ocurrencias(tree,2)` devuelve 2
`ocurrencias(tree,null)` devuelve 3
`ocurrencias(tree,7)` devuelve 0

Ejercicio 131: Expresión matemática almacenada en un árbol

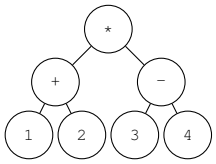
medio

Implementar en Java el método

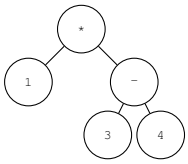
```
public static String toStringExp (BinaryTree<String> tree)
```

que toma como parámetro un árbol binario `tree` y devuelve un `String` con la expresión matemática en notación «infija» almacenada en el árbol. Será necesario poner paréntesis en cada subexpresión para evitar posibles problemas con la precedencia de operadores. Cada nodo del árbol contendrá un elemento de la expresión en forma de String, ya sea operador u operando. El árbol `tree` no será **`null`** ni podrá contener elementos **`null`**. Se asume que `tree` contendrá expresiones matemáticas correctas y todos los nodos del árbol que contengan un operador tendrán dos hijos. Se **recomienda** implementar el método **de forma recursiva**.

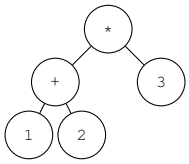
A continuación se muestran algunos ejemplos de árboles y los resultados devueltos por el método `toStringExp`:



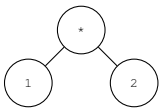
`"((1+2)*(3-4))"`



`"(1*(3-4))"`



`"((1+2)*3)"`



`"(1*2)"`

Ejercicio 132: Corrigiendo método `hasHeapProperty`

medio

Queremos implementar un método en Java que compruebe si un árbol binario cumple la propiedad que caracteriza a los montículos (*heaps*):

> *El elemento contenido en cada nodo es mayor o igual que el elemento contenido en su padre, si tiene padre.*

Se quiere, además, que sea razonablemente eficiente y, en particular, que no realice operaciones innecesarias para determinar si se cumple o no la propiedad. Asumimos que los nodos del árbol contienen objetos de la clase `Integer`, que ningún objeto contenido en un nodo es **`null`**. El árbol al que aplicamos el método puede estar vacío, en cuyo caso debemos devolver que cumple la propiedad. Nos proporcionan el siguiente código **erróneo** para resolver este problema:

```
1  // Returns @\textbf{true}@ iff @\textbf{tree}@ satisfies the heap-order property
2  public static boolean hasHeapProperty (BinaryTree<Integer> tree) {
3    return tree.isEmpty() || hasHeapProperty(tree,tree.root());
4  }
5
6  public static boolean hasHeapProperty (BinaryTree<Integer> tree,
7                                         Position<Integer> node) {
8
9    if (node.element() < tree.parent(node).element() && tree.parent(node) != null) {
10     return false;
11   }
12
13   boolean res = true;
14   if (tree.hasLeft(node)) {
15     res = hasHeapProperty (tree,tree.left(node));
16   }
17   if (tree.hasRight(node)) {
18     hasHeapProperty (tree,tree.right(node));
19   }
20   return res;
21 }
```

Determinar qué cambios son necesarios para que el código devuelva el resultado correcto, no se lance ninguna excepción y para hacer el código más eficiente evitando que se realicen operaciones innecesarias. Para contestar debéis indicar el número de línea en el que está el problema y como quedaría la línea para resolverlo.

Ejercicio 133: Implementando `hojasEnRango`

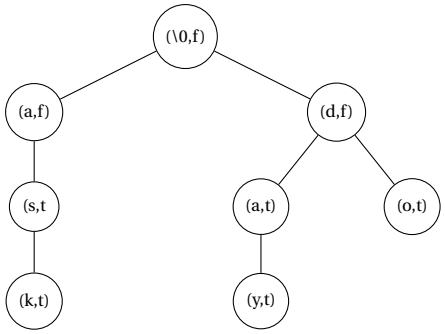
fácil

implementar el método

```
public static void printWordsInTrie (Tree<Pair<Character,Boolean>> tree)
```

que recibe como parámetro un árbol `Tree<Pair<Character,Boolean>>`, que representa un *trie*, y debe imprimir todas las palabras (el orden de impresión no es relevante) contenidas en el árbol. Cada nodo del árbol tiene un elemento (*carácter*, *booleano*) y una palabra p está en el trie si se cumplen dos condiciones: (1) hay una secuencia de nodos desde la raíz cuyos caracteres concatenados forman la palabra p y (2) el nodo final de esa secuencia tiene el valor booleano **`true`**. Tanto `tree` como su contenido nunca serán **`null`**.

Por ejemplo, el método imprimirá las palabras `as`, `ask`, `day`, `do` para el árbol que se muestra a continuación. El orden en el que las imprima no es relevante y podéis usar `System.out.println()` para imprimir.



NOTA: El carácter contenido en el nodo raíz siempre será '\0', que se corresponde con el carácter *vacío* por lo que se podrá concatenar a un `String` sin que esto añada ningún carácter al `String`. Asimismo, tened en cuenta que los objetos de tipo `Character` ser pueden concatenar a un `String` utilizando el operador "+", de la misma forma que se utiliza para concatenar dos Strings. Por ejemplo, dado el siguiente código:

```
Character a = 'a';
Character b = '\0';
String s = "Prueba: ";
String s = s + a + b;
```

el `String` s contendra `"Prueba: a"`.

Ejercicio 134: Implementando hojasEnRango

fácil

Implementar en Java el método:

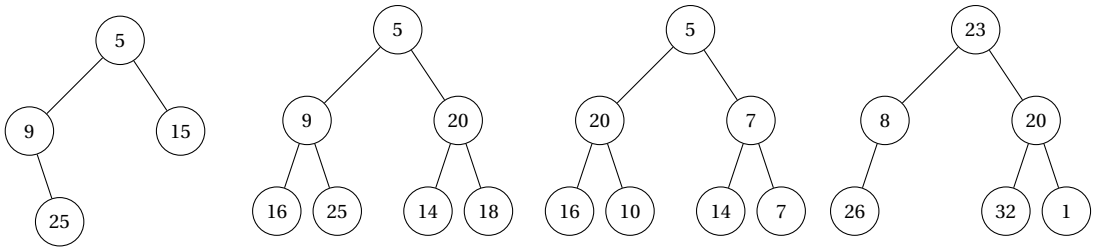
```
PositionList<Integer> hojasEnRango (BinaryTree<Integer> tree, int a, int b)
```

que recibe como parámetro un árbol binario `tree` y un rango definido por `a` y `b`, y devuelve una lista con todos los nodos hoja que se encuentran dentro del rango `[a,b]`. El rango recibido siempre será correcto, es decir, $a < b$ y un valor `x` está dentro del rango `[a,b]` si $a \leq x \leq b$. El árbol `tree` nunca contendrá elementos **`null`**. El árbol `tree` podría ser **`null`**, en cuyo caso el método debe lanzar la `IllegalArgumentException`. En caso de que `tree` esté vacío, el método debe devolver la lista vacía.

Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Dado el rango `a = 15, b = 25` y los siguientes árboles, las listas resultantes deben contener los siguientes elementos.

NOTA: el orden de los elementos en la lista resultado no es relevante.



Ejercicio 135: Método `sumaNodos2Hijos`

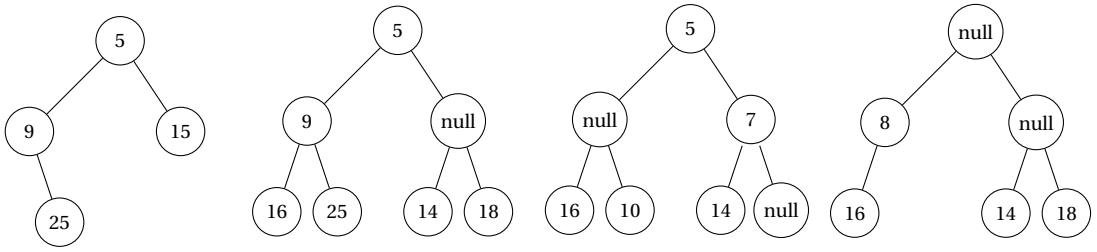
medio

Implementar en Java el método:

```
static int sumaNodos2Hijos (BinaryTree<Integer> tree)
```

que recibe como parámetro un árbol binario `tree` y devuelve la suma de los elementos contenidos en los nodos del árbol que tienen 2 hijos. El árbol `tree` podrá contener elementos null, que no computan en la suma. El árbol `tree` podría ser **`null`**, en cuyo caso el método debe lanzar la `IllegalArgumentException`. En caso de que `tree` esté vacío, el método debe devolver `0`.

Dados los siguientes árboles, el resultado sería:



Colas con prioridad / montículos / *heaps*

Ejercicio 136: Complejidad operaciones de colas con prioridad

medio

Una posible implementación de colas con prioridad consiste en utilizar un array cuyos índices (de `0` a un `M` dado) representen las prioridades, y con una cola FIFO en cada posición del array para almacenar los elementos que tienen como prioridad la correspondiente al índice de la posición (i-ésima posición del array cola con todos los elementos que tienen prioridad i). Se pide:

(a) Una cola con prioridad puede almacenar elementos distintos que sin embargo tengan la misma prioridad. Di qué implementaciones de colas con prioridad de entre las dos siguientes, devolverán el elemento con mínima prioridad más antiguo en la estructura, al invocar la operación `first()`:

- La implementación con array de colas FIFO aquí sugerida.
- Una implementación basda en un montículo.

(b) Sí cuál sería el orden de complejidad en el peor caso de las operaciones `enqueue`, `dequeue` y `first` para una cola con prioridad implementada de esta forma. Asumir que la complejidad de las operaciones de inserción de un elemento, borrado y consulta del primer elemento en una cola FIFO `Q`, tienen complejidades respectivas $O(m)$, $O(1)$ y $O(1)$, siendo m el número de elementos en `Q`.

Ejercicio 137: Dibujar montículo

medio

Sea `q` una cola con prioridad implementada mediante un montículo. Las claves son números enteros y los valores son caracteres. La implementación de la cola utiliza un vector («array») de tamaño 5 para almacenar el montículo. Inicialmente `q` está vacía y a continuación se realiza la siguiente secuencia de operaciones:

```
q.enqueue(5,'A');
q.enqueue(4,'B');
q.enqueue(7,'I');
q.enqueue(1,'D');
q.dequeue();
```

(a) Dibujar el árbol (casi) completo del montículo resultante.

(b) Dibujar el contenido del vector resultante.

Ejercicio 138: Dibujar montículo

medio

Se tiene una cola con prioridad implementada mediante un montículo. Las claves son números enteros y los valores son caracteres. Inicialmente la cola está vacía.

Dibujar el árbol (casi)completo del montículo resultante de insertar la siguiente secuencia de entradas (3,'A'), (7,'I'), (5,'B'), (8,'F'), (2,'G'), (4,'C') seguido del borrado de la entrada de mayor prioridad.

Ejercicio 139: Dibujar montículo

medio

Sea `q` una cola con prioridad implementada mediante un montículo. Las claves son números enteros y los valores son caracteres. La implementación de la cola utiliza un vector («array») de tamaño 5 para almacenar el montículo. Inicialmente `q` está vacía y a continuación se realiza la siguiente secuencia de operaciones:

```
q.enqueue(4,'B');
q.enqueue(7,'I');
q.enqueue(3,'A');
q.enqueue(1,'D');
q.dequeue();
```

(a) Dibujar el árbol (casi)completo del montículo resultante.

(b) Dibujar el contenido del vector resultante.

Ejercicio 140: Implementando una PriorityQueue

medio

El siguiente método implementa el borrado de la entrada con menor clave (con mayor prioridad) de una cola con prioridad implementada mediante una lista almacenada en la variable de instancia `entries`:

```
/** Removes and returns an entry with minimum key. */
public Entry<K,V> dequeue() throws EmptyPriorityQueueException {
  if (entries.isEmpty())
    throw new EmptyPriorityQueueException("priority queue is empty");
  else
    return entries.remove(entries.first());
}
```

La variable de instancia `entries` guarda un objeto lista de clase `PositionList<Entry<K,V>>`. A la luz de éste código:

(a) Determinar si `entries` es una lista ordenada o una lista arbitraria. Razone brevemente su respuesta.

(b) Después de varias pruebas con la implementación, se ha decidido que es mejor invertir el orden de las claves asociadas a los valores. Un programador sugiere que la forma más sencilla de acomodar el cambio es sustituyendo en el código anterior la línea

```
return entries.remove(entries.first());
```

por la siguiente:

```
return entries.remove(entries.last());
```

Comente brevemente la adecuación de ésta propuesta.

Ejercicio 141: ¿Qué es un *heap*?

fácil

¿Qué es un *heap*?

Ejercicio 142: Dibujar montículo

medio

La siguiente figura muestra el contenido del vector de una implementación mediante montículo de una cola con prioridad.

null	(2,X)	(4,B)	(3,Y)	(7,A)	null	null	null

(a) Dibujar el árbol (casi) completo representado por el vector.

(b) Dar la secuencia de entradas que se obtendrían al invocar el método `dequeue()` consecutivamente hasta vaciar la cola con prioridad.

Ejercicio 143: Dibujar montículo

medio

La siguiente figura muestra el contenido del vector de un montículo. Se muestran las claves almacenadas, obviando los valores:

3	3	3	7	3			

(a) Dibujar el árbol (casi)completo que forma el montículo:

(b) Dibujar el árbol resultante de invocar `dequeue` dos veces consecutivas sobre el árbol anterior.

(c) Dibujar el vector correspondiente al árbol de la respuesta anterior.

Ejercicio 144: Dibujar montículo

medio

La siguiente figura muestra el contenido del vector de un montículo. Se muestran las claves almacenadas, obviando los valores:

5	6	10	7	11	11		

(a) Dibujar el árbol (casi)completo que forma el montículo:

(b) Dibujar el árbol resultante de invocar `dequeue` dos veces consecutivas sobre el árbol anterior.

(c) Dibujar el vector correspondiente al árbol de la respuesta anterior.

Ejercicio 145: Dibujar montículo

medio

La siguiente figura muestra las claves (obviaremos los valores) de las entradas almacenadas en el «array» de un montículo que implementa una cola con prioridad:

2	5	8	7	10	null	null	null

(a) Dibujar el árbol resultante de insertar en el montículo anterior una entrada cuya clave sea 6.

(b) Dibujar el árbol resultante de invocar `dequeue` sobre el montículo dado como respuesta en el apartado anterior.

Ejercicio 146: Dibujar montículo

medio

La siguiente figura muestra las claves (obviaremos los valores) de las entradas almacenadas en el «array» de un montículo que implementa una cola con prioridad:

5	6	10	7	null	null	null	null

(a) Dibujar el árbol resultante de insertar en el montículo anterior una entrada cuya clave sea 3.

(b) Dibujar el árbol resultante de invocar `dequeue` sobre el montículo dado como respuesta en el apartado anterior.

Ejercicio 147: Dibujar montículo

medio

Se tiene una cola con prioridad vacía implementada mediante un montículo sobre la que se llevan a cabo las siguientes operaciones:

- Se insertan de forma consecutiva las siguientes entradas (sólo mostramos las claves de tipo `Integer`, obviaremos los elementos): 5, 4, 3, 3, 9, 8, 6.
- Inmediatamente después se borra la entrada con clave mínima.

La cola con prioridad utiliza internamente el comparador estándar de enteros. Dibujar los árboles (casi)completos que conforman el montículo en cada paso (añadir la entrada al árbol, «up-heap», «down-heap», borrar la entrada del árbol), etiquetando los árboles de los pasos de «up-heap» con la letra **U** y los árboles de los pasos de «down-heap» con la letra **D**.

Ejercicio 148: Dibujar montículo

medio

En una cola con prioridad implementada mediante un montículo se insertan de forma consecutiva las siguientes entradas (sólo mostramos las claves de tipo `Integer`, podemos ignorar los elementos): 4, 3, 1, 1, 8, 7, 5, e inmediatamente después se borra la entrada con clave mínima.

La cola con prioridad utiliza internamente un comparador estándar para `Integer` de forma que una clave k_1 tiene mayor o igual prioridad que k_2 cuando $k_1 \leq k_2$.

(a) Dibujar el árbol (casi)completo de cada paso (añadir la entrada al árbol, borrar la entrada del árbol, «up-heap» y «down-heap») etiquetando los árboles de los pasos de «up-heap» con la letra **U** y los árboles de los pasos de «down-heap» con la letra **D**.

(b) Dibujar el vector resultante del árbol final (tras el borrado de la entrada con clave mínima).

Ejercicio 149: Buscar los elementos con clave mínima de una lista

Implementar en Java el método:

```
public static <K,V> PositionList<Entry<K,V>> minList(PriorityQueue<Entry<K,V>> pq)
```

que toma como parámetro un objeto cola con prioridad. Si la cola es null o vacía entonces el método devuelve una nueva lista vacía. En otro caso el método devuelve como resultado una nueva lista con todas las entradas de la cola que tienen la clave mínima, colocándolas en la lista en el mismo orden de izquierda a derecha en el que se sacan de la cola. Por ejemplo, si se usan como claves números enteros (`K` es `Integer`) y como valores

caracteres (`V` es `Character`), suponiendo que la cola tiene las entradas `(5,'X')`, `(5,'Y')`, `(5,'Z')`, `(8,'X')`, `(8,'Z')`, `(10,'Y')` entonces el método devuelve una nueva lista con las entradas `(5',X')`, `(5,'Y')`, `(5,'Z')` en ese orden. El método puede modificar la cola `pq`.

Ejercicio 150: Ordenando Alumnos

fácil

Se tiene la siguiente clase `Alumno`:

```
class Alumno {
  private Integer dni;
  private String apellidos;

  public Alumno(Integer dni, String apellidos) {
    this.dni = dni;
    this.apellidos = apellidos;
  }

  public Integer getDni() { return dni; }
  public String getApellidos() { return apellidos };
}
```

(a) Implementar el método

```
public static Alumno [] ordenarAlumnos (Alumno [] alumnos)
```

que toma como parámetro un array de objetos de tipo `Alumno` que no será **`null`** y cuyos elementos tampoco serán **`null`** y devuelve un *nuevo* array del mismo tamaño que el array `alumnos` con los alumnos ordenados en orden creciente en función de su DNI. El método debe devolver el resultado en un nuevo array y **no debe modificar** el parámetro `alumnos`. Se recomienda utilizar una cola con prioridad usando el interfaz `PriorityQueue<K,V>` para ordenar los elementos. Puede asumirse que se dispone de la clase `SortedListPriorityQueue<K,V>` que implementa el interfaz `PriorityQueue<K,V>`.

(b) Implementar el método `compareTo` en la clase `Alumno` para que implemente el interfaz `Comparable<Alumno>` estableciendo un orden creciente de acuerdo al número del DNI. La definición de la clase alumno ahora sería:

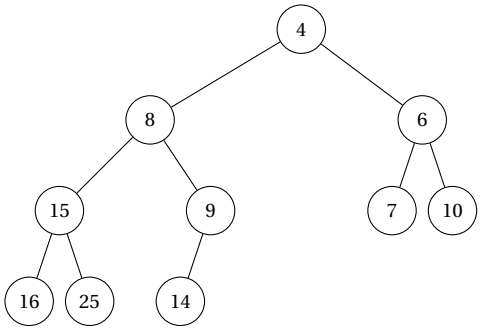
```
class Alumno implements Comparable<Alumno> {...}
```

Sólo es necesario implementar el método `compareTo`, no es necesario copiar en la solución el resto de la clase `Alumno`.

Ejercicio 151: Encolar y desencolar en un montículo

fácil

Dado el siguiente montículo (*heap*)



Dibujar el estado final del montículo después de ejecutar las siguientes operaciones sobre dicho montículo.

(a) `enqueue(5)`

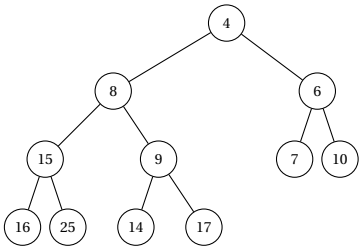
(b) `dequeue()`

NOTA: **NO** apliquéis `dequeue()` sobre el montículo resultante de ejecutar `enqueue(5)`. Tanto `enqueue(5)` como `dequeue()` se aplican sobre el montículo tal como está en el dibujo.

Ejercicio 152: Encolar y desencolar en un montículo

fácil

Dado el siguiente montículo (*heap*)



Dibujar el estado final del montículo después de ejecutar las siguientes operaciones.

(a) `enqueue(3)`

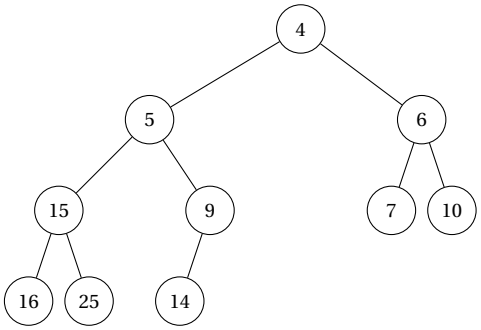
(b) `dequeue()`

IMPORTANTE: NO apliquéis `dequeue()` sobre el montículo resultante de ejecutar `enqueue(3)`. Tanto `enqueue(3)` como `dequeue()` se aplican sobre el montículo tal como está en el dibujo.

Ejercicio 153: Encolar y desencolar en un montículo

fácil

Dado el siguiente montículo (*heap*)



Dibujar el estado final del montículo después de ejecutar las siguientes operaciones.

(a) `enqueue(4)`

(b) `dequeue()`

IMPORTANTE: NO apliquéis `dequeue()` sobre el montículo resultante de ejecutar `enqueue(4)`. Tanto `enqueue(4)` como `dequeue()` se aplican sobre el montículo tal como está en el dibujo.

Maps

Ejercicio 154: Requisitos de los Maps

fácil

Las funciones finitas («maps») y los diccionarios imponen un requisito mínimo sobre las claves: debe de poder comprobarse su igualdad. Indicar dos formas de realizar este requisito cuando la clase de las claves es `Integer`.

Ejercicio 155: Implementando un Map

medio

Se tiene una implementación de funciones finitas («maps») que usa una lista desordenada para almacenar las entradas:

```
class MapList<K,V> implements Map<K,V> {
  public NodePositionList<Entry<K,V>> lista;
  public MapList()  { lista = new NodePositionList<Entry<K,V>>(); }
  public int size() { return lista.size(); }
  public boolean isEmpty() { return lista.isEmpty(); }
  public V get(K k) {
    /** COMPLETAR **/
  }
  ...  // resto de metodos
}
```

Implementar el matodo `get` usando el iterador de la lista para recorrerla. El interfaz `Map<K,V>` está disponible en el Apéndice.

Ejercicio 156: Contar apariciones

medio

Implementar en Java el método

```
public static Map<Character,Integer> contarApariciones(String texto)
```

que devuelve un `Map<Character,Integer>` cuyas claves serán cada uno de los caracteres que aparecen en el parámetro `texto` y el valor asociado a cada clave será el número de veces que aparece el carácter en `texto`. El parámetro `texto` no será **`null`**. Se dispone de la clase `HashTableMap<K,V>` que implementa el interfaz `Map<K,V>` y que cuenta con el constructor `HashTableMap<K,V>()` para crear un `Map` vacío.

NOTA: La clase `String` dispone del método `length()` para obtener el número de caracteres del `String` y el método **`char`** `charAt(int i)` que devuelve el carácter que ocupa la posición `i` del `String` siendo 0 la primera posición y `length()-1` la última posición válida.

Por ejemplo, dado el `texto = "En un lugar de La Mancha"` la llamada al método `contarApariciones(texto)` debe devolver un `Map` que contenga los siguiente pares (clave,valor): ('h',1), (' ',5), ('M',1), ('u',2), ('E',1), ('e',1), ('r',1), ('g',1), ('l',1), ('d',1), ('L',1), ('a',4), ('n',3), ('c',1), que indica que, por ejemplo, el carácter 'n' aparece 3 veces en `texto` o que el carácter 'e' aparece 1 vez.

Ejercicio 157: Aumentando el precio de productos

fácil

Implementar en Java el método

```
public static void aumentarPrecio (Map<String,Double> precios,
                                   String prod, Double coste)
```

que, dada una tabla de precios `precios`, un producto `prod` y un coste `coste`, debe aumentar en dicha tabla el precio del producto teniendo en cuenta lo siguiente: (1) si el producto ya está en la tabla aumentar su precio con el valor de `coste`; (2) si el producto no está en la tabla, se establece como precio del producto el valor de `coste`. La clave del map es el identificador del producto y su valor asociado es el precio. Por ejemplo, el map `precios = <"1",10>,<"3",13>` indica que tenemos dos productos, uno con identificador `"1"` y precio `10` y otro producto con identificador `"3"` y precio `13`.

Por ejemplo, dado `precios=[<"1",10>,<"3",13>]`, la llamada a `aumentarPrecio(precios,"1",2)` debe dejar en el map `precios=[<"1",12>,<"3",13>]`.

O dado, `precios=[<"2",14>,<"8",17>]` la llamada `aumentarPrecio(precios,"5",12)` deberá dejar `precios=[<"2",14>,<"8",17>, <"5",12>]`.

Ejercicio 158: Contando prácticas

fácil

Dada la clase `Alumno` de la pregunta anterior, se dispone de un listado en el que están mezcladas todas las entregas de todos los alumnos que han entregado las prácticas individuales. En dicho listado un mismo alumno podrá aparecer múltiples veces, tantas como prácticas haya entregado.

Implementar en Java el método

```
public Map<Integer,Integer> contarPracticas (PositionList<Alumno> lentregas)
```

que recibe una lista con las entregas de los alumnos y devuelve un objeto de tipo `Map<Integer,Integer>` cuya clave será el DNI del alumno y cuyo valor será el número de entregas realizadas por dicho alumno. La lista `lentregas` no contendrá elementos **`null`** ni los atributos de los alumnos contenidos en la lista serán **`null`**. Se dispone de la clase `HashTableMap<K,V>` que implementa el interfaz `Map<K,V>` y que cuenta con el constructor `HashTableMap<K,V>()` para crear un `Map` vacío.

Por ejemplo, dada `lista=[Alumno(4,"a"),Alumno(2,"b"),Alumno(0,"c"),Alumno(4,"a")]`, el método `contarPracticas(lista)` devolverá un `Map` con los siguientes pares ⟨clave,valor⟩: ⟨2,1⟩, ⟨0,1⟩, ⟨4,2⟩, que indica que el alumno con DNI 2 ha entregado 1 práctica, que el alumno con DNI 0 ha entregado 1 práctica y que el alumno con DNI 4 ha entregado 2 prácticas.

Ejercicio 159: Invertir Map

fácil

Decimos que un *map* es la inversa de otro cuando todos los elementos de los pares `<k,v>` del original aparecen en el inverso pero con `v` como nueva clave y `k` como nuevo valor – es decir, como pares `<v,k>`. Para que un map sea invertible, todos sus valores deben ser distintos y no nulos.

Implementar en Java el método:

```
static Map<String,String> invertir (Map<String,String> map)
```

que recibe como parámetro un `Map<String, String>` invertible y devuelve el map inverso.

Por ejemplo, dado `map = [<"k1","v1">,<"k2","v2">,<"k3","v3">]`, el método debe devolver un nuevo map con los pares `[<"v1","k1">,<"v2","k2">,<"v3","k3">]`.

Ejercicio 160: Ejecutando un método

fácil

Dado el siguiente método:

```
public static void desconocido (String s) {
  Map<Character,Integer> map = new HashTableMap<Character,Integer>();
  for(int i = 0; i < s.length(); i ++) {
    Integer n = map.get(s.charAt(i));
    if (n == null) {
      map.put(s.charAt(i),1);
    }
    else {
       map.put(s.charAt(i),n+1);
    }
  }

  Iterator<Entry<Character,Integer>> it = map.entries();
  while (it.hasNext()) {
    Entry<Character,Integer> entry = it.next();
    System.out.println("(" + entry.getKey() + " - " + entry.getValue() + "), ");
  }
}
```

Mostrar una posible salida por consola de la ejecución del siguiente método si se ejecuta la siguiente llamada: `desconocido("imprimo?");`

NOTA: Recordad que el método `s.charAt(i)` devuelve el carácter que ocupa la posición `i` en `s` (siendo `s` un `String`.

Ejercicio 161: Implementando Set

fácil

`Set<E>` es un interfaz que define las operaciones de un conjunto finito de elementos de clase `E`. Un conjunto es una colección de elementos en la que no existe orden y en la que no hay elementos repetidos. implementar la clase `MiSet<E>` que implemente el interfaz `Set<E>`, mostrado a continuación, **utilizando internamente un atributo de tipo `Map<E,E>`** (no está permitido añadir otros atributos):

```
public interface Set<E> {
    /** Devuelve true si el conjunto está vacío y false en caso contrario. */
    public boolean isEmpty();

    /** Devuelve el número de elementos del conjunto. */
    public int size();

    /** Añade el elemento ``elem'' al conjunto. Devuelve true si el
      * elemento ya estaba contenido en el conjunto y false en caso
      * contrario. En caso de que ``elem'' sea null el método debe lanzar
      * la excepción IllegalArgumentException y no se añade al conjunto. */
    public boolean add(E elem);

    /** Elimina ``elem'' del conjunto. Devuelve true si el elemento
      * estaba en el conjunto y ha sido eliminado y false en caso
      * contrario. En caso de que ``elem'' sea null el método debe
      * lanzar la excepción IllegalArgumentException. */
    public boolean remove(E elem);

    /** Devuelve true si el elemento ``o'' está en conjunto. En caso
      * de que ``elem'' sea null el método debe devolver false. */
    public boolean contains(Object o);
```

```
    /** Devuelve todos los elementos contenidos en el conjunto en una
      * estructura Iterable. El orden de los elementos en el Iterable no
      * es relevante. */
    public Iterable<E> getElements();

    /** Devuelve un nuevo conjunto intersección, que contiene los elementos
      *  que están simultáneamente en los conjuntos ``set'' y ``this''.
      *  El conjunto ``set'' nunca será null */
    public Set<E> intersection(Set<E> set);
}
```

Se dispone de la clase `HashTableMap<K,V>`, que implementa el interfaz `Map<K,V>` y que dispone de un constructor sin parámetros para crear un `Map` vacío.

IMPORTANTE!! En la descripción de los interfaces entregada en el examen hay una errata en el interfaz `Map<E,E>`: Falta el método:

```
/** Returns true if the map contains an entry with the key argument,
 * and false otherwise. */
public boolean containsKey(Object key) throws InvalidKeyException;
```

Ejercicio 162: Implementando `getAlumnosEnRango`

fácil

Implementar en Java el método:

```
PositionList<String> getAlumnosEnRango (Map<String,Integer> map, int a, int b)
```

que recibe un map `map` cuyas claves son los números de matrícula de los alumnos y el valor asociado a cada clave es la nota obtenida por el alumno. El método debe devolver una lista con los números de matrícula cuyas notas se encuentren dentro del rango `[a,b]`. El rango recibido siempre será correcto, es decir, $a < b$ y un valor `x` está dentro del rango `[a,b]` si $a \leq x \leq b$. El orden de los elementos en la lista resultado no es relevante. Los valores contenidos en `map` nunca serán **`null`**.

Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía.

Por ejemplo, dado `map = [<"123",5>,<"456",8>,<"789",3>,<"345",10>]`, la llamada `getAlumnosEnRango(map,0,5)` devolverá `["123","789"]`,`getAlumnosEnRango(map,9,10)` devolverá `["345"]` y `getAlumnosEnRango(map,6,7)` devolverá `[]`.

Ejercicio 163: Implementando `ordenarPorNota`

fácil

Implementar en Java el método:

```
static <E> PositionList<String> ordenarPorNota (Map<String,Integer> map)
```

que recibe como parámetro un `Map<String,Integer>` que tiene como clave el DNI de un estudiante y como valor la nota obtenida en un examen, y debe devolver una **nueva** lista que contenga los DNIs de los alumnos ordenados de forma ascendente según las notas que han obtenido. El `Map` no contendrá notas con valor **`null`** y el map de entrada nunca será **`null`**. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía. Asimismo, se dispone de la clase `HeapPriorityQueue<K,V>`, que implementa el interfaz `PriorityQueue<K,V>` y que dispone de un constructor sin parámetros para crear una cola con prioridad vacía.

Por ejemplo, la llamada `ordenarPorNota([<"123",7>,<"456",9>,<"789",5>])` deberá devolver `["789","123","456"]` y la llamada `ordenarPorNota([<"123",10>,<"456",3>,<"789",5>])` deberá devolver `["456","789","123"]`.

Ejercicio 164: Implementando `ordenarDescPorApariciones`

fácil

Implementar en Java el método:

```
static <E> PositionList<E> ordenarDescPorApariciones (Map<String,Integer> map)
```

que recibe como parámetro un `Map<E,Integer>` que tiene como clave un objeto cualquiera y como valor el número de veces que aparece, y debe devolver una **nueva** lista que contenga los elementos contenidos en el `Map` ordenador de forma **descendiente** según el número de apariciones. El `Map` no contendrá elementos **`null`** y el map de entrada nunca será **`null`**. El valor de las entradas en el map siempre será correcto, es decir, no será **`null`** y será un valor positivo. Se dispone de la clase `NodePositionList<E>`, que implementa el interfaz `PositionList<E>` y que dispone de un constructor sin parámetros para crear una lista vacía. Asimismo, se dispone de la clase `HeapPriorityQueue<K,V>`, que implementa el interfaz `PriorityQueue<K,V>` y que dispone de un constructor sin parámetros para crear una cola con prioridad vacía.

Por ejemplo, la llamada `ordenarDescPorApariciones([<a,7>,<b,9>,<c,5>])` deberá devolver `[b,a,c]` y la llamada `ordenarDescPorApariciones([<a,10>,<b,3>,<c,5>])` deberá devolver `[a,c,b]`.

Complejidad y análisis de memoria

Ejercicio 165: Orden de órdenes de complejidad

medio

Ordena (con los operadores $\subset$ y/o $=$) los siguientes conjuntos: $O(\log_e n)$, $O(1)$, $O(n!)$, $O(\log_5 n)$, $O(n)$, $O(n^4)$, $O(n^2)$, $O(n \log n)$, $O(2^n)$.

Ejercicio 166: Complejidad de algoritmo de suma

medio

Considera el siguiente algoritmo para sumar los valores almacenados en un array `A` de números enteros cuyo tamaño `n` es una potencia de dos: se crea un array `B` cuyo tamaño es la mitad del tamaño del array `A`, y se asigna `B[i] = A[2i] + A[2i+1]` para `i = 0, 1, ... (n/2)-1`. Si `B` tiene tamaño 1 se devuelve `B[0]`, en otro caso se reemplaza `A` con `B` y se repite el proceso ¿Cuál es el orden de complejidad del algoritmo?

Ejercicio 167: Complejidad de memoria y tiempo de `IndexList` disperso

medio

Decimos que un `IndexList<E>` es disperso si en la mayor parte de sus posiciones el valor almacenado es **`null`**. Podemos implementar un `IndexList` disperso `A` con una cadena enlazada, de forma que cada nodo de la misma almacena un par `(i,e)` correspondiente a una de las posiciones `A[i]` que almacenan un valor `e` no nulo.

(a) ¿De qué orden es el requisito de memoria de esta implementación? (Expresado en notación $O()$).

(b) Compara la complejidad de la operación `set(i,e)` dependiendo de que los pares se almacenen en la cadena en orden creciente de índices o no.

Ejercicio 168: `IndexedList` con cadena doblemente enlazada

fácil

Si implementasemos `IndexedList<E>` utilizando una cadena doblemente enlazada, ¿cuál sería la complejidad en el peor caso de la operación `add(i, e)`?

Ejercicio 169: Complejidad de inserción en `ArrayIndexedList` con copia

medio

Almacenamos una colección de elementos en una implementación de `ArrayIndexList` de forma que tras cada inserción se traslada la colección a un nuevo array cuyo tamaño es exactamente el número de elementos de la misma ¿Cual es la complejidad de realizar n operaciones consecutivas de inserción de un elemento partiendo de una colección vacía?

Ejercicio 170: Complejidad de máximo de un vector de enteros

medio

Sea la siguiente función del cálculo del máximo de un array de enteros `a_int` empleando la técnica de Divide y Vencerás. En la llamada inicial `indice_inicio` es 0, e `indice_final` es `n-1`, donde `n` es el número de elementos de `a_int`.

```
public static int maximoArrayDyV (int[] a_int,
                                  int indice_inicio,
                                  int indice_final) {
   int indice_medio;
   if (indice_inicio == indice_final)  // Caso basico: se resuelve directamente
      return a_int[indice_inicio];
   else {
     indice_medio = (indice_inicio + indice_final) / 2;
     return java.lang.Math.max(
                 maximoArrayDyV (a_int, indice_inicio, indice_medio),
                 maximoArrayDyV (a_int, indice_medio + 1, indice_final));
   }
}
```

¿Cuál es su complejidad asintótica?

Ejercicio 171: Complejida de multiplicación de matrices

fácil

El siguiente código multiplica dos matrices cuadradas y revuelve una tercera matriz cuadrada:

```
/*
 * Multiply two square matrices together.
 */
public static int[][] multiply(int[][] mat1, int[][] mat2) {
  int size = mat1.length;
  int[][] result = new int[size][size];

  // Do the multiplication
  for (int i = 0; i < size; i++)
    for (int j = 0; j < size; j++)
      for (int k = 0; k < size; k++)
        result[i][j] += mat1[i][k] * mat2[k][j];

  return result;
}
```

¿Cuál es la complejidad de este código, en función del número de filas (o columnas) n de las matrices de entrada?

Ejercicio 172: Ordenación de medidas de complejidad

medio

Enumerar de menor a mayor las siguientes medidas asintóticas de complejidad:

A. $O(\log n + \log n)$
B. $O(n \log n)$
C. $O((\log n)^2)$
D. $O(n \times n)$
E. $O(4n + \log n)$

Ejercicio 173: Complejidad de comprobacion de una propiedad

fácil

Sea `m` un método que toma como argumento un objeto de la clase `ArrayIndexedList<Integer>` y que itera 5 veces sobre los elementos del *array* para encontrar 5 enteros que cumplen 5 propiedades diferentes. La complejidad de determinar si un entero arbitrario cumple una de las 5 propiedades es $O(1)$. Indicar cuál de las siguientes es la medida de complejidad de `m` en el caso peor, donde n es, en todos los casos, el número de elementos almacenados en el *array*:

(a) $O(5 * n)$

(b) $O(5n+5)$

(c) $O(n)$

(d) $O(n \times n)$

Ejercicio 174: Ordenación de medidas de complejidad

medio

Dadas las siguientes medidas asintóticas de complejidad:

A: $O(n^2 \log n)$

B: $O(2^{\log n^2})$

C: $O(\log n)$

D: $O(n^3)$

E: $O(\log 2^{(n+1)})$

Enumerarlas de menor a mayor. (Recuérdese que $2^{\log n} = n$ y $\log 2^n = n$)

Ejercicio 175: Complejidad de un videojuego

medio

Se tiene un videojuego de carreras de coches en el que pueden correr hasta 20 coches. La siguiente posición de *un* coche es una función que toma como parámetros las posiciones actuales de todos los coches. La complejidad de dicha función es constante. El videojuego dispone de un algoritmo que toma un vector con las posiciones de *todos* los coches y calcula la siguiente posición de cada coche. Indicar la complejidad en caso peor de dicho algoritmo.

Ejercicio 176: Complejidad de creación de una lista ordenada

medio

Se tiene un método que toma como parámetro un vector o «array» de elementos y devuelve un objeto lista de clase `NodePositionList` con los elementos del vector ordenados de forma creciente. Indicar la complejidad en el caso peor de dicho método, razonando la respuesta, y suponiendo que no se altera ni copia el vector original.

Ejercicio 177: Complejidad de búsqueda de una lista en otra

para pensar

El siguiente método toma como parámetro una lista de objetos de clase `Alumno` e indica si todos ellos aparecen en una lista de alumnos matriculados `listaMatriculados` cuyo tamaño es fijo (dicha lista no puede modificarse):

```
public bool matriculados(PositionList<Alumno> lista) {
  Iterator<Alumno> it = lista.iterator();
  boolean all = true;
  while (it.hasNext() && all)
    all = all && listaMatriculados.member(it.next());
  return all;
}
```

Indicar de entre las siguientes posibilidades la complejidad en caso peor del método `matriculados`, donde n el tamaño de `lista`:

(a) $O(1)$

(b) $O(n^2)$

(c) $O(n)$

(d) $O(n \times m)$, con m el tamaño de `listaMatriculados`.

Ejercicio 178: Complejidad de `show` (ejercicio 82)

fácil

Indicar razonadamente la complejidad en tiempo en el caso peor que tendrá el método `show` de la pregunta 82 cuando el iterador es de clase `NonNullElementIterator<Integer>`.

Ejercicio 179: Complejidad de `next` (ejercicio 83)

fácil

Razonar la complejidad en caso peor del método `next()` del ejercicio 83. Las respuestas sin razonar serán calificadas con 0 puntos.

Ejercicio 180: Complejidad de 2 métodos

fácil

Indicar la complejidad de los siguientes métodos:

```
<E> boolean m1 (PositionList<E> l, E e) {
  boolean res = false;
  Position<E> cursor = l.first();
  if (cursor!=null) {
    res = cursor.element().equals(e);
  }
  return res;
}
```

```
void m2 (PositionList<E> l) {
  Position<E> cursor = l.first();
  while (cursor!=null) {
    Position<E> cursor2 = l.first();
    while (cursor2!=null) {
      cursor2 = l.next(cursor2);
    }
    cursor = l.next(cursor);
  }
}
```

Ejercicio 181: Complejidad de 2 métodos

fácil

Indique la complejidad de los siguientes métodos:

```java
void m1 (PositionList<E> l) {
  Position<E> cursor = l.first();
  while (cursor!=null) {
    maux(l);
    cursor = l.next(cursor);
  }
}
void maux (PositionList<E> l) {
  Position<E> cursor = l.first();
  while (cursor!=null) {
    cursor = l.next(cursor);
  }
}
```

```java
boolean m2 (PositionList<E> l, E e) {
  Position<E> cursor = l.first();
  while (cursor!=null &&
      !cursor.element().equals(e)) {
    cursor = l.next(cursor);
  }
  return cursor!=null;
}
```

Ejercicio 182: Complejidad de 2 métodos

fácil

Indicar la complejidad de los métodos `m1` y `m2`:

```java
void m1 (IndexedList<E> l) {
  maux(l, 0);
}
void maux (IndexedList<E> l,
           int c) {
  if (c >= l.size()) {
    return;
  }
  maux(l,c+1);
}
```

```java
void m2 (PositionList<E> l, E e) {
   Position<E> c = l.first();
   while (c!=null) {
      Position<E> c2 = l.last();
      while (c2 != null) {
        c2 = l.prev(c2);
      }
      c = l.next(c);
   }
}
```

Ejercicio 183: Complejidad de 2 métodos

fácil

Indicar la complejidad de los siguientes métodos:

```java
boolean m1 (PositionList<E> l, E e) {
  boolean res = false;
  Position<E> cursor = l.first();
  if (cursor!=null) {
    res = cursor.element().equals(e);
  }
  return res;
}
```

```java
void m2 (PositionList<E> l) {
  Position<E> cursor = l.first();
  while (cursor!=null) {
    Position<E> cursor2 = l.first();
    while (cursor2!=null) {
      cursor2 = l.next(cursor2);
    }
    cursor = l.next(cursor);
  }
}
```

Ejercicio 184: Complejidad de 2 métodos

fácil

Indicar la complejidad de los métodos `m1` y `m2`:

```
void m1 (PositionList<E> l) {
  maux(l, l.first());
}
void maux (PositionList<E> l,
           Position<E> c) {
  if (c == null) {
    return;
  }
  maux(l,l.next(c));
}
```

```
void m2 (PositionList<E> l, E e) {
   Position<E> c = l.first();
   while (c!=null) {
      c = l.next(c);
   }
   Position<E> c2 = l.first();
   while (c2!=null) {
      c2 = l.next(c2);
   }
}
```

Ejercicio 185: Complejidad de 2 métodos

fácil

Indicar la complejidad de los métodos `method1` y `method2`:

```
<E> void method1 (PositionList<E> list) {
  Position<E> c = list.first();
  while (c != null && c.element() != null) {
    System.out.println("B1: " + c.element());
    c = list.next(c);
  }

  while (c != null) {
    System.out.println("B2: " + c.element());
    c = list.next(c);
  }
}
```

```
<E> int method2 (IndexedList<E> l) {
  int i = l.size();
  int counter = 0;
  while (i > 0) {
    counter ++;
    i = i / 2;
  }
  return counter;
}
```

Ejercicio 186: Complejidad de 2 métodos

fácil

Indicar la complejidad de los métodos `method1` y `method2`:

```java
<E> void method1 (PositionList<E> list) {
  Position<E> c = list.first();
  while (c != null) {
    Position<E> c2 = list.first();
    while (c2 != null) {
      ....println("E: " + c2.element());
      c2 = list.next(c2);
    }
    c = list.next(c);
  }

}
```

```java
<E> int method2 (IndexedList<E> l) {
  int counter = 0;
  for (int i = 0; i < l.size(); i++) {
    int j = l.size();
    while (j > 0) {
      counter ++;
      j = j / 2;
    }
  }
  return counter;
}
```

Ejercicio 187: Complejidad de 2 métodos

fácil

Indicar la complejidad de los métodos `method1` y `method2`:

```java
<E> boolean method1 (PositionList<E> l) {
  Position<E> c = l.first();
  boolean allNoNull = true;
  while (c != null && allNoNull) {
    allNoNull = c.element() != null;
    c = l.next(c);
  }
  return allNoNull;
}
```

```java
<E> int method2 (PositionList<E> l){
  int counter = 0;
  for (int i = l.size();
       i > 0;
       i = i/2) {
    counter ++;
  }
  return counter;
}
```

Ejercicio 188: Complejidad de 2 métodos

fácil

Indicar la complejidad de los métodos `method1` y `method2`:

```java
<E> boolean method1 (PositionList<E> l) {
  Position<E> c = l.first();
  boolean hayNull = false;
  while (c != null && !hayNull) {
    hayNull = c.element() == null;
    c = l.next(c);
  }
  return hayNull;
}
```

```java
int method2 (int[] a){
  int counter = 0;
  for (int i=0;i<a.length;i++) {
    for (int j=0;j<a.length;j++) {
      counter++;
    }
  }
  return counter;
}
```

Grafos

Ejercicio 189: Corrigiendo `isReachable`

medio

Se pretende implementar en Java el método

```
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to)
```

que devuelve **`true`** si desde el vértice `from` se puede alcanzar el vértice `to` y **`false`**, en caso contrario. Nos proporcionan el siguiente código **erróneo** para resolver este problema:

```
1  public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
2                                           Vertex<V> from,
3                                           Vertex<V> to) {
4    Set<Vertex<V>> visited = new HashTableMapSet<Vertex<V>>();
5    return isReachable(g, from, to, visited);
6  }
7
8  public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
9                                           Vertex<V> from,
10                                          Vertex<V> to,
11                                          Set<Vertex<V>> visited ) {
12
13   if (from == to) {
14     return false;
15   }
16
17   visited.add(from);
18   boolean reachable = false;
19   Iterator<Edge<E>> it = g.edges(from).iterator();
20   while (it.hasNext()) {
21     Vertex<V> other = g.opposite(from, it.next());
22     if (!visited.contains(other)) {
23       isReachable(g, other, to, visited);
24     }
25   }
26   return reachable;
27 }
```

Determinar qué cambios son necesarios para que el código devuelva el resultado correcto, no se lance ninguna excepción sea más eficiente, evitando realizar operaciones innecesarias. Para contestar debéis indicar el número de línea en el que está el problema y como quedaría la línea para resolverlo.

Ejercicio 190: Terminando `getVerticesAlcanzables`

medio

Se pretende implementar en Java el método `getVerticesAlcanzables` que, dado un grafo `g` y un vértice `n`, devuelve el conjunto de vértices alcanzables desde `n`, es decir, que se puede encontrar un camino en el grafo desde `n` hasta dichos vértices. Se proporciona la siguiente parte del código:

```
1  static <V,E> Set<Vertex<V>> getVerticesAlcanzables (UndirectedGraph<V, E> g,
2                                                      Vertex<V> n) {
3    Set<Vertex<V>> visited = new HashTableMapSet<V>();
4    getVerticesAlcanzablesRec(g,n,visited);
5    return visited;
6  }
7  static <V,E> void getVerticesAlcanzablesRec (UndirectedGraph<V, E> g,
8                                               Vertex<V> n,
9                                               Set<Vertex<V>> visited ) {
10
11   // COMPLETAR ESTE METODO
12 }
```

Completar el código del método `getVerticesAlcanzablesRec` para que implemente la funcionalidad indicada.

NOTA: Para añadir elementos al conjuto `visited` podéis usar el método `visited.add(n)`, que añade el vértice `n` al conjunto `visited`.

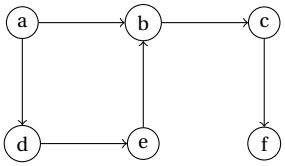
Ejercicio 191: Terminando `isReachableInNSteps`

medio

Se quiere implementar en Java el método

```
isReachableInNSteps(DirectedGraph<V, E> g, Vertex<V> from, Vertex<V> to, int n)
```

que, dado un grafo *dirigido* `g`, dos vértices `from` y `to`, y un valor numérico `n`, devuelve **`true`** si el nodo `to` es alcanzable desde el nodo `from` atravesando, como máximo, `n` aristas. Los valores de los parámetros nunca serán **`null`** y siempre serán valores correctos. Por ejemplo, dado el siguiente grafo `g`, las llamadas a `isReachableInNSteps` deben devolver:




```
isReachableInNSteps(g,a,c,1) -> false
isReachableInNSteps(g,a,c,2) -> true
isReachableInNSteps(g,a,b,2) -> true
isReachableInNSteps(g,b,a,2) -> false
```

Se dispone del siguiente código, que contiene errores:

```
1   public static <V,E> boolean isReachableInNSteps (DirectedGraph<V, E> g,
2                                                    Vertex<V> from, Vertex<V> to,
3                                                    int n) {
4 |\label{error}|    return isReachableInNStepsError(g, from, to, n);
5   }
6
7   public static <V,E> boolean isReachableInNStepsRec (DirectedGraph<V, E> g,
8                                                       Vertex<V> from, Vertex<V> to,
9                                                       int n) {
10   if (from == to) {
11     return true;
12   }
13   if  (n >= 0) {
14     return false;
15   }
16   boolean reachable = false;
17   Iterator<Edge<E>> it = g.edges(from).iterator();
18   while (it.hasNext()) {
19     reachable = isReachableInNStepsRec(g, g.endVertex(from), to, n, visited);
20   }
21   return reachable;
22 }
```

Indicar cuáles son las líneas erróneas y cuál sería el código correcto de las líneas que contienen errores. Por ejemplo:

```
L|\ref{error}| -> return isReachableInNSteps(g, from, to, n);
```

El código dado tiene 4 líneas con errores (además del error de la línea **??** ya indicado) y las líneas erróneas podrían tener como máximo 2 errores.

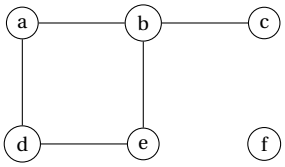
Ejercicio 192: Corrigiendo `isReachable`

medio

Se quiere implementar en Java el método

```
isReachable (UndirectedGraph<V, E> g, Vertex<V> from, Vertex<V> to)
```

que, dado un grafo *no dirigido* `g`, dos vértices `from` y `to` devuelve **`true`** si el nodo `to` es alcanzable desde el nodo `from`. Los valores de los parámetros nunca serán **`null`** y siempre serán valores correctos. El grafo `g` nunca contendrá elementos **`null`**. Por ejemplo, dado el siguiente grafo `g`, las llamadas a `isReachable` deben devolver:




```
isReachable(g,a,c) -> true
isReachable(g,c,a) -> true
isReachable(g,a,f) -> false
isReachable(g,f,d) -> false
```

Se dispone del siguiente código, que contiene errores:

```
1  public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
2                                           Vertex<V> from,
3                                           Vertex<V> to) {
4   Set<Position<?>> visited = new HashTableMapSet<>();
5  |\label{error}| isReachable(g, from, to, visited);
6  }
7  public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
8                                           Vertex<V> from,
9                                           Vertex<V> to,
10                                          Set<Position<?>> visited ) {
11
12   if (from == to) {
13     return true;
14   }
15   if (visited.contains(from)) {
16     return true;
17   }
18
19   visited.remove(from);
20   boolean reachable = false;
21   Iterator<Edge<E>> it = g.edges(from).iterator();
22   while (it.hasNext()) {
23     reachable = isReachable(g, g.opposite(from, to), to, visited);
24   }
25   return reachable;
26 }
```

Indicar cuáles son las líneas erróneas y cuál sería el código correcto de las líneas que contienen errores. Por ejemplo:

```
L|\ref{error}| -> return isReachable(g, from, to, visited);
```

El código dado tiene 4 líneas con errores (además del error de la línea **??** ya indicado).

Ejercicio 193: Implementar `reachableNodesWithGrade`

fácil

Se pretende implementar en Java el método `reachableNodesWithGrade` que, dado un grafo *dirigido* `g`, un vértice `v` y un grado `grade`, devuelve una lista que contiene los vértices **alcanzables** desde `v` **que tienen grado saliente igual a `grade`**. El orden de la lista resultado no es relevante.

```
  static <V,E> PositionList<Vertex<V>>
    reachableNodesWithGrade (DirectedGraph<V, E> g,
                             Vertex<V> v, int grade) {

    Set<Vertex<V>> visited  = new HashTableMapSet<>();
    PositionList<Vertex<V>> res = new NodePositionList<>();
    reachableNodesWithGrade(g, v, grade, visited, res);
    return res;
  }

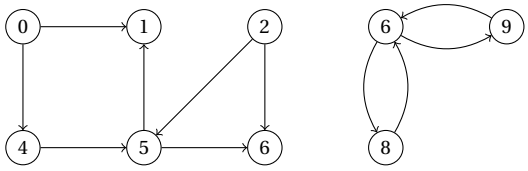
  public static <V,E> void reachableNodesWithGrade (DirectedGraph<V, E> g,
                                                    Vertex<V> v,
                                                    int grade,
                                                    Set<Vertex<V>> visited,
                                                    PositionList<Vertex<V>> res) {

  // COMPLETAR ESTE METODO
}
```

Completar el código del método `reachableNodesWithGrade` para que implemente la funcionalidad indicada. El grafo nunca será **`null`** ni contendrá vértices con elementos **`null`**.

Nota: Para añadir elementos al conjunto `visited` podéis usar el método `visited.add(v)`, que añade el vértice `v` al conjunto `visited`. Para comprobar si un elemento está presente en `visited` se puede ejecutar `visited.contains(v)` que devuelve **`true`** si `v` está presente, y **`false`** si no.

Por ejemplo, dado el grafo que sigue, el método pedido, empezando en el nodo `v` que contiene `0`, debe devolver la lista con los elementos reseñados sin que importe el orden de los elementos en la lista:



`reachableNodesWithGrade(g,v,2) = [0,5]`

Se pretende implementar en Java el método `reachableNodesWithGrade` que, dado un grafo *no dirigido* `g`, un vértice `v` y un número de pasos `steps`, devuelve el número de vértices distintos **alcanzables** desde `v` **en igual o menos pasos que el valor de `steps`**.

```
  public static <V,E> int reachableNodesWithGrade (UndirectedGraph<V, E> g,
                                                   Vertex<V> v,
                                                   int steps) {

    Set<Vertex<?>> visited = new HashTableMapSet<>();
    visited.add(v);
    reachableNodesWithGrade(g,v,visited,steps);
    return visited.size();
```

```
  }

  public static <V,E> void reachableNodesWithGrade (UndirectedGraph<V, E> g,
                                                    Vertex<V> v,
                                                    Set<Vertex<?>> visited,
                                                    int steps) {

  // COMPLETAR ESTE METODO
}
```

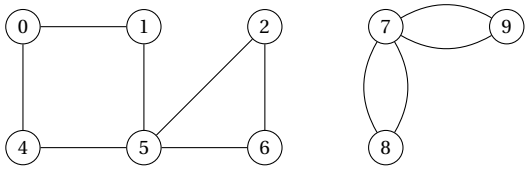
Ejercicio 194: Implementar `numReachableInSteps`

fácil

Completar el código del método `numReachableInSteps` para que implemente la funcionalidad indicada. El grafo nunca será **`null`** ni contendrá vértices con elementos **`null`**. El vértice `v` nunca será **`null`** y siempre será un vértice contenido en el grafo. El valor de `steps` siempre será mayor o igual que 0.

Nota: Para añadir elementos al conjunto `visited` podéis usar el método `visited.add(v)`, que añade el vértice `v` al conjunto `visited`. Para comprobar si un elemento está presente en `visited` se puede ejecutar `visited.contains(v)` que devuelve **`true`** si `v` está presente, y **`false`** si no.

Por ejemplo, dado el siguiente grafo `g`, el método pedido deberá devolver lo siguiente. Notad que `v(x)` hace referencia al nodo que contiene el valor `x`:



`numReachableInSteps(g,v(0),1)=3`, dado que alcanza los vértices `[0,1,4]`
`numReachableInSteps(g,v(0),0)=1`, dado que alcanza los vértices `[0]`
`numReachableInSteps(g,v(0),2)=4`, dado que alcanza los vértices `[0,1,4,5]`
`numReachableInSteps(g,v(0),3)=6`, dado que alcanza los vértices `[0,1,2,4,5,6]`
`numReachableInSteps(g,v(8),1)=2`, dado que alcanza los vértices `[7,8]`.

Soluciones propuestas

Ejercicio 1 (solución propuesta)

```
for (int j=size-1 ; j>=i ; j--)
  A[j+1] = A[j] ;
A[j] = e ;
size++ ;
```

Ejercicio 2 (solución propuesta)

```
for (int i=r; i<size-1; i++)
  A[i] = A[i+1];
size--;
```

Ejercicio 3 (solución propuesta)

(c)

Ejercicio 4 (solución propuesta)

```
public int [] reverse(int [] v) {
  int [] r = null;
  if (v != null) {
    r = new int[v.length];
    if (v.length > 0)
        for (int i = 0; i < v.length; i++)
            r[i] = v[v.length-1-i];
  }
  return r;
}
```

La solución anterior devuelve **null** cuando v es **null**, pasándole el problema al código que haya invocado reverse. Las soluciones que hayan tratado **null** de otra forma, o hayan asumido que v no es null las consideramos válidas, pues el enunciado no dice nada sobre **null**.

Ejercicio 5 (solución propuesta)

```
public static boolean iguales (Integer [] arr1, Integer [] arr2) {
  if (arr1 == arr2 || arr1.length == 0 && arr2.length == 0) return true;
  if (arr1.length != arr2.length) return false;
  int i;
  for (i = 0; i < arr1.length && eqNull(arr1[i],arr2[i]); i++)
    ;
  return i == arr1.length;
}
private static boolean eqNull (Integer i1, Integer i2) {
  return i1 == i2 || (i1 != null && i1.equals(i2));
}
```

Ejercicio 6 (solución propuesta)

```
public static <E> boolean esInverso (E [] arr1, E [] arr2) {
  if (arr1.length != arr2.length)
    return false;
  int i, j;
  for (i = 0, j = arr2.length-1;
       i < arr1.length && eqNull(arr1[i],arr2[j]);
       i++, j--)
     ;

  return i == arr1.length;
}

  public static <E> boolean eqNull (E o1, E o2) {
    return o1 == o2 || o1!=null && o1.equals(o2);
  }
```

Ejercicio 7 (solución propuesta)

```
public class ColorHex implements ColorRGB {
  private String hex;
  ...
  public int getRed()    { return getRedFromHex(this.hex);   }
  public int getGreen()  { return getGreenFromHex(this.hex); }
  public int getBlue()   { return getBlueFromHex(this.hex); }
  public String getHex() { return this.hex; }
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o instanceof ColorRGB) {
       ColorRGB c = (ColorRGB) o;
       return this.getHex().equals(c.getHex());
    } else
      return false;
  }
}
public class ColorInts implements ColorRGB {
  private int r, g, b;
  ...
  public int getRed()    { return this.r; }
  public int getGreen()  { return this.g; }
  public int getBlue()   { return this.b; }
  public String getHex() { return getHexValue(this.r,this.g,this.b); }
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o instanceof ColorRGB) {
       ColorRGB c = (ColorRGB) o;
       return this.getHex().equals(c.getHex());
    } else
      return false;
  }
}
```

Ejercicio 8 (solución propuesta)

```
public class TimeHMS implements Time {

  private int hours;
  private int mins;
  private int secs;
  ...
  public int getHours() { return hours; }
  public int getMins() { return mins; }
  public int getSecs() { return secs; }
  public int timeInSeconds(){ return hours*3600+mins*60+secs; }
}

public class TimeSec implements Time {

  private int totalSecs;
  ...
  public int getHours() { return totalSecs / 3600; }
  public int getMins() { return (totalSecs % 3600) / 60; }
  public int getSecs() { return totalSecs % 60; }
  public int timeInSeconds () { return totalSecs; }
  public boolean equals(Object o){
    if (o == this) {return true; }
    if (o instanceof Time){
      Time t = (Time) o;
      return t.timeInSeconds() == this.timeInSeconds();
    } else {
      return false;
    }
  }
}
```

Ejercicio 9 (solución propuesta)

```
public int compare(Iterator<E> it1, Iterator<E> it2) {
  int i = 0;
  while (it1.hasNext() && it2.hasNext() &&
         (i = compElem.compare(it1.next(),it2.next())) == 0)
    ;
  if (i==0 && (it1.hasNext() || it2.hasNext()))
    i = it1.hasNext() ? 1 : -1 ;
  return i;
}
```

Ejercicio 10 (solución propuesta)

(a)
```
public int compareTo(Dato o) {
  return this.valor - o.valor;
}
```

(b)
```
public static boolean estaOrdenado (Dato [] datos) {
  int i = 0;
  while (i < datos.length-1 && datos[i].compareTo(datos[i+1]) <= 0)
    i++;
  return i==datos.length-1;
}

public static boolean estaOrdenado (Dato [] datos) {
  int i;
  for (i = 0;
    i < datos.length-1 && datos[i].compareTo(datos[i+1]) <=0; i++) ;
  return i == datos.length-1;
}
```

Ejercicio 11 (solución propuesta)

```
static <E extends Comparable<E>> boolean estaOrdenada (PositionList<E> list) {
  if (list.size() <= 1) return true;
  Position<E> cursor = list.next(list.first());
  while (cursor != null &&
         list.prev(cursor).element().compareTo(cursor.element()) <= 0){
    cursor = list.next(cursor);
  }
  return cursor == null;
}
```

Ejercicio 12 (solución propuesta)

(a)
1. Donde aparece `list.last()` deberemos escribir `list.first()`
2. No
3. Sustituir `list.first()` por `list.last()` y `list.prev(cursor)` por `list.next(cursor)`.

(b)
```
public static <E> boolean iguales(PositionList<E> l1,PositionList<E> l2 ){
      IteradorListas<E> it1=new IteradorListas<E>(l1);
      IteradorListas<E> it2=new IteradorListas<E>(l2);
      int size=l1.size();
      if (size!=l2.size()) return false;
      boolean resul=true;
      if (size>0){
          while (it1.hasNext())
             if (!(it1.next().equals(it2.next()))) return false;
      }
      return resul;
  }
```

(c) $O(n)$ siendo n la longitud de una lista (en realidad de ambas, porque el peor caso es cuando las listas son iguales y por tanto tienen la misma longitud). Se recorren dos listas de longitud n ejecutando operaciones que tienen complejidad constante. Recordemos que $O(2n) = O(n)$.

Ejercicio 13 (solución propuesta)

```
public IndexedList<E> take(int n, IndexedList<E> list) {
  IndexedList<E> res = new ArrayIndexedList<E>() ; // lista resultado
  int tope = n<=0 ? 0 : Math.min(n,list.size()) ;
  for (int i = 0 ; i < tope ; i++)
      res.add(i,list.get(i)) ;
}
```

Ejercicio 14 (solución propuesta)

```
for (Iterator<Position<E>> it = list.positions();
     n>0 && it.hasNext(); n--)
  rlist.addLast(it.next().element());
```

Ejercicio 16 (solución propuesta)

La idónea es `NodePositionList` pues su método `addFirst(p)` inserta el último pedido `p` al principio de la lista con complejidad $O(1)$ en el caso peor, mientras que el método `add(0,p)` de `ArrayIndexedList` inserta el último pedido `p` al principio de la lista con complejidad $O(n)$ en el caso peor.

Ejercicio 17 (solución propuesta)

```
public void append(PositionList<E> list) {
  Position<E> cursor = list.first();
  while (cursor != null) {
     addLast(cursor.element());
     cursor = list.next(cursor);
  }
```

Ejercicio 18 (solución propuesta)

Solución que inserta `e1` antes de la primera ocurrencia de `e2`:

```
public void addBeforeElement(PositionList<E> list, E e1, E e2) {
  if (!list.isEmpty()) {
    Position<E> p = list.first();
    for (int n = 1; n < list.size() && !p.element().equals(e2); n++)
      p = list.next(p);
    // Aqui p es o la posicion con e2 o la ultima posicion.
    if (p.element().equals(e2)) list.addBefore(p,e1);
  }
}
```

Solucion que inserta `e1` antes de todas las ocurrencias de `e2`:

```
public void addBeforeElement(PositionList<E> list, E e1, E e2) {
  for (Position<E> p : list.positions())
    if (p.element().equals(e2)) list.addBefore(p,e1)
}
```

Ejercicio 19 (solución propuesta)

1. La invocación `list.first()` lanzará una excepción si la lista parámetro es null.
2. La invocación `list.next(p)` lanzará una `NullPointerException` cuando `p` sea el último nodo de la lista, lo cual sucede si `e` no está en la lista.

Ejercicio 20 (solución propuesta)

```
public void replace(IndexedList<E> list, E e1, E e2) {
  for (int i = 0; i < list.size(); i++)
    if (list.get(i).equals(e1)) {
      list.set(i,e2);
    }
}
```

Ejercicio 21 (solución propuesta)

```
  public void reverse(ArrayIndexedList<E> list) {
    if (!list.isEmpty()) {
      for ( int start = 0, int end = list.size()-1;
            end-start > 0;
            start++, end-- ) {
        E temp = list.get(start);
        list.set(start,list.get(end));
        list.set(end,temp);
      }
    }
  }
```

Ejercicio 22 (solución propuesta)

```
public PositionList<E> hastaNull(PositionList<E> list) {
  NodePositionList<E> r  = new NodePositionList<E>();
  Iterator<E> it = list.iterator();
  E e;
  while (it.hasNext() && (e = it.next()) != null) r.addLast(e);
  return r;
}
```

Ejercicio 23 (solución propuesta)

```
  public int numoc(E elem, PositionList<E> list) {
    int n = 0 ;
    Iterator<E> it = list.iterator() ;
    while (it.hasNext()) {
      E e = it.next();
      if (e != null && e.equals(elem)) n++ ;
    }
    return n;
  }
```

Ejercicio 24 (solución propuesta)

Solución basada en iteradores:

```
public boolean equals(PositionList<E> list) {
  if (this == list) return true ;
  if (list == null) return false ;
  if (this.size() != list.size()) return false ;
  if (this.size() == 0) return true ; // both lists empty
  // At this point both lists non-empty and same size
  Iterator<E> itthis = this.iterator() ;
  Iterator<E> itlist = list.iterator() ;
  boolean equal = true ;
  E thisElem, listElem ;
  while (itthis.hasNext() && equal) {
    thisElem = itthis.next() ;
    listElem = itlist.next() ;
    if (thisElem == null && listElem != null ||
        thisElem != null && listElem == null)
      equal = false ;
    else
      equal = thisElem.equals(listElem) ;
  }
  return equal ;
}
```

Solución sin iteradores:

```
public boolean equals(PositionList<E> list) {
  if (this == list) return true ;
  if (list == null) return false ;
  if (this.size() != list.size()) return false ;
  if (this.size() == 0) return true ; // both lists empty
  // At this point both lists non-empty and same size
  Position<E> pthis = this.first() ;
  Position<E> plist = list.first() ;
  boolean equal = true ;
  E thisElem, listElem ;
  while (pthis != null && equal) {
    thisElem = pthis.element() ;
    listElem = plist.element() ;
    if (thisElem == null && listElem != null ||
        thisElem != null && listElem == null)
      equal = false ;
    else
      equal = thisElem.equals(listElem) ;
    pthis = pthis == this.last() ? null : this.next(pthis) ;
    plist = plist == list.last() ? null : list.next(plist) ;
  }
  return equal ;
}
```

Ejercicio 25 (solución propuesta)

```
public void filter(int i, PositionList<Integer> list) {
  Iterator<Integer> it = list.iterator();
  Integer cur;
  while (it.hasNext()) {
    cur = it.next();
    if (cur != null && cur == i) it.remove();
  }
}
```

Ejercicio 26 (solución propuesta)

```
  public void addFirst(E element) {
    Node<E> newNode = new Node<E>(header, header.getNext(), element);
    header.getNext().setPrev(newNode);
    header.setNext(newNode);
    numElts++;
  }
```

Ejercicio 27 (solución propuesta)

Solución que usa el `remove` del iterador:

```
  public void removeAll(PositionList<E> list, E e) {
    Iterator<E> it = list.iterator();
    while (it.hasNext())
      if (it.next().equals(e)) it.remove();
  }
```

Solución que usa un «snapshot»:

```
  public void removeAll(PositionList<E> list, E e) {
    for (Position<E> p : list.positions())
      if (p.element().equals(e)) list.remove(p);
  }
```

Ejercicio 28 (solución propuesta)

Solución que recorre la lista `l1` y usa `member`:

```
public void removeList(PositionList<E> l1, PositionList<E> l2)
  if (!l1.isEmpty() && !l2.isEmpty()) {
    Position<E> cursor = l1.first();
    Position<E> tmp;
    while (cursor != null) {
      tmp    = cursor;
      cursor = cursor == l1.last() ? null : l1.next(cursor);
      if (member(tmp.element(),l2)) l1.remove(tmp);
    }
  }
}
```

Solución que recorre la lista `l2` y no usa `member`

```
public void removeList(PositionList<E> l1, PositionList<E> l2) {
  if (!l1.isEmpty() && !l2.isEmpty()) {
    Position<E> cursor1, cursor2;
    for ( cursor2 = l2.first();
          cursor2 != null
          cursor2 = cursor2 == l2.last() ? null : l2.next(cursor2) )
     {
       for ( cursor1 = l1.first();
             cursor1 != null && !cursor2.element().equals(cursor1.element());
             cursor1 = cursor1 == l1.last() ? null : l1.next(cursor1) )
         ;
       if (cursor1 != null) l1.remove(cursor1);
       if (l1.isEmpty()) cursor2 = null;
     }
}
```

Ejercicio 29 (solución propuesta)

(a) Es invocado por los métodos que toman parámetros de tipo `Position<E>`.

(b) Comprueba que `p` referencia un nodo válido: que `p != null`, que `p` es de clase `Node<E>`, que tras el «downcasting» los enlaces anterior y siguiente no son `null`, y devuelve el nodo. Si no se cumple alguno de los anteriores entonces lanza la excepción.

Ejercicio 30 (solución propuesta)

Solución con «for-each»:

```
PositionList<Integer> r = new NodePositionList<Integer>();
for (PositionList<Integer> listaDeEnteros : listaDeListas)
  for (Integer entero : listaDeEnteros)
    r.addLast(entero);
return r;
```

Solución con iteradores explícitos:

```
PositionList<Integer> r = new NodePositionList<Integer>();
Iterator<PositionList<Integer>> itLL = listaDeListas.iterator();
while (itLL.hasNext()) {
  Iterator<Integer> itL = itLL.next().iterator();
  while (itL.hasNext())
    r.addLast(itL.next());
}
return r;
```

Damos también a modo ilustrativo las soluciones con control de null:

```
if (listaDeListas == null) return null;
else {
  PositionList<Integer> r = new NodePositionList<Integer>();
  for (PositionList<Integer> listaDeEnteros : listaDeListas)
    if (listaDeEnteros != null)
      for (Integer entero : listaDeEnteros)
        if (entero != null)
          r.addLast(entero);
  return r;
}
```

```
if (listaDeListas == null) return null;
else {
  PositionList<Integer> r = new NodePositionList<Integer>();
  for (Iterator<PositionList<Integer>> itLL = listaDeListas.iterator();
       itLL.hasNext();
      ) {
    PositionList<Integer> listaDeEnteros = itLL.next();
    if (listaDeEnteros != null) {
      for (Iterator<Integer> itL = listaDeEnteros.iterator(); itL.hasNext(); ) {
        Integer entero = itL.next();
        if (entero != null)
          r.addLast(entero);
      }
    }
  }
  return r;
}
```

Ejercicio 31 (solución propuesta)

Solución que primero determina el orden y luego borra:

```
public static <E>  void delete( PositionList<E> list,
                                Position<E> pos1,
                                Position<E> pos2) {
  if (pos1 == pos2) list.remove(pos1);
  else {
    Position<E> aux = pos1;
    while (aux != null && aux != pos2)
      aux = list.next(aux);

    Position<E> from  = aux != null ? pos1 : pos2;
    Position<E> to    = list.next(aux != null ? pos2 : pos1);

    while (from != to) {
      Position<E> del = from;
      from = list.next(from);
      list.remove(del);
    }
  }
}
```

Solución que determina el orden y borra a la vez:

```
public static <E>  void delete( PositionList<E> list,
                                Position<E> pos1,
                                Position<E> pos2) {
  if (pos1 == pos2) list.remove(pos1);
  else {
    Position<E> cur = list.first();
    int found       = 0;
    while (found < 2) {
      if (cur == pos1 || cur == pos2)
        found++;
      Position<E> nxt = list.next(cur);
      if (found > 0)
        list.remove(cur);
      cur = nxt;
    }
  }
}
```

Ejercicio 32 (solución propuesta)

Solución con **while**:

```
public static <E> E get(PositionList<E> list, int n) {
  if (list == null || n < 0 || n > list.size()-1) return null;
  Position<E> cursor = list.first();
  while (n > 0) {
    cursor = list.next(cursor);
    n--;
  }
  return cursor.element();
}
```

Solución con **for**:

```
public static <E> E get(PositionList<E> list, int n) {
  if (list == null || n < 0 || n > list.size()-1) return null;
  Position<E> cursor;
  for (cursor = list.first(); n > 0; cursor = list.next(cursor), n--)
    ;
  return cursor.element();
}
```

Ejercicio 33 (solución propuesta)

En la penúltima línea se le pasa un nodo de `list2` a `list1` y por tanto se lanzará la excepción `IllegalArgumentException`. (Los nodos de las listas tienen un atributo `owner` que almacena el «hashcode» de la lista a la que pertenece. Todos los métodos de la lista que toman nodos como parámetro, entre ellos `addBefore`, invocan `checkNode` que comprueba que el nodo pasado como parámetro tiene el mismo «hashcode» que el nodo `header` de la lista.)

Ejercicio 34 (solución propuesta)

```
public static PositionList<Integer> flatNub (PositionList<Integer> [] arr) {
  PositionList<Integer> res = new NodePositionList<Integer>();
  if (arr.length == 0) return res;  // podria quitarse esta linea
  for (int i = 0; i < arr.length; i++) {
    Position<Integer> cursor = arr[i].first();
    while (cursor != null) {
      if (!member(cursor.element(),res))
        res.addLast(cursor.element());
      cursor = arr[i].next(cursor);
    }
  }
  return res;
}
```

Ejercicio 35 (solución propuesta)

```
public static PositionList<Integer> join (PositionList<Integer> l1,
                                          PositionList<Integer> l2) {
  PositionList<Integer> res = new NodePositionList<Integer>();
  Position<Integer> c1 = l1.first();
  Position<Integer> c2 = l2.first();

  while (c1 != null && c2 != null) {
    if (c1.element() <= c2.element()) {
      res.addLast(c1.element());
      c1 = l1.next(c1);
    }
    else {
      res.addLast(c2.element());
      c2 = l2.next(c2);
    }
  }
  while (c1!=null) {
    res.addLast(c1.element());
    c1 = l1.next(c1);
  }
  while (c2!=null) {
    res.addLast(c2.element());
    c2 = l2.next(c2);
  }
  return res;
}
```

Ejercicio 36 (solución propuesta)

```
PositionList<Integer> intercalar (PositionList<Integer> l1,
                                  PositionList<Integer> l2) {
  PositionList<Integer> res = new NodePositionList<Integer>();
  Position<Integer> c1 = l1.first();
  Position<Integer> c2 = l2.first();

  while (c1 != null && c2 != null) {
    res.addLast(c1.element());
    c1 = l1.next(c1);
    res.addLast(c2.element());
    c2 = l2.next(c2);
  }
  while (c1!=null) {
    res.addLast(c1.element());
    c1 = l1.next(c1);
  }
  while (c2!=null) {
    res.addLast(c2.element());
    c2 = l2.next(c2);
  }
  return res;
}
```

Ejercicio 37 (solución propuesta)

```
public static <E> int getNumApariciones(PositionList<E> lista, E elem) {
  if (lista == null || lista.isEmpty())
    return 0;
  return getNumApariciones(lista, elem, lista.first());
}

private static <E> int getNumApariciones(PositionList<E> lista,
                                         E elem,
                                         Position<E> cursor) {
  if (cursor == null) return 0;    /* caso base */

  return (elem.equals(cursor.element()) ? 1 : 0) +
          getNumApariciones(lista,elem,lista.next(cursor));
}

/* Con recursion de cola*/

public static <E> int getNumAparicionesAcum(PositionList<E> lista,
                                            E elem) {
  if (lista == null || lista.isEmpty()) return 0;

  return getNumAparicionesAcum(lista, elem, lista.first(), 0);
}

private static <E> int getNumAparicionesAcum(PositionList<E> lista,
                                             E elem,
                                             Position<E> cursor,
                                             int acum) {
  if (cursor == null) return acum;    /* caso base */

  return getNumAparicionesAcum(lista, elem, lista.next(cursor),
                               (elem.equals(cursor.element()) ? 1 : 0) + acum);
}
```

Ejercicio 81 (solución propuesta)

```
  private void moveCursor () {
    while (cursor != null &&
           (cursor.element()== null || cursor.element() <= 0)) {
       cursor = list.next(cursor);
    }
}
```

Ejercicio 39 (solución propuesta)

```
PositionList<Integer> getElementosImpares (PositionList<Integer> lista) {
  if (lista == null) {
    throw new IllegalArgumentException();
  }

  PositionList<Integer> res = new NodePositionList<Integer>();

  for (Integer e: lista) {
    if (e != null && (e%2 == 1) && !member(res,e)) {
      res.addLast(e);
    }
  }
  return res;
}
```

Ejercicio 40 (solución propuesta)

```
public static <E> void eliminarConsecIguales (PositionList<E> lista) {
  if (lista.size() < 2) return;

  Position<E> actual = lista.first();
  Position<E> cursor = lista.next(actual);

  while(cursor != null) {
    if (eqNull(cursor.element(),actual.element())) {
      lista.remove(cursor);
    }
    else {
      actual = lista.next(actual);
    }
    cursor = lista.next(actual);
  }
}

private static boolean eqNull(Object o1, Object o2) {
  return o1 == null && o2 == null || o1 != null && o1.equals(o2);
}
```

Ejercicio 41 (solución propuesta)

```
static <E> PositionList<E> barajar (PositionList<E> list) {
  if (list == null) throw new IllegalArgumentException ();
  Position<E> cini = list.first();
  Position<E> cfin = list.last();
  PositionList<E> res = new NodePositionList<E>();
  while (cini != cfin) {
    res.addLast(cini.element());
    res.addLast(cfin.element());
    cini = list.next(cini);
    cfin = list.prev(cfin);
  }
  res.addLast(cini.element());
  return res;
}
```

Ejercicio 42 (solución propuesta)

```
static <E> PositionList<E> copiarHastaElem (PositionList<E> list,
                                            E elem) {

  PositionList<E> nueva = new NodePositionList<E>();

  Position<E> cursor = list.first();
  while(cursor != null && !eqNull(cursor.element(),elem)) {
    nueva.addLast(cursor.element());
    cursor = list.next(cursor);
  }

  return nueva;
}

private static boolean eqNull(Object o1, Object o2) {
  return o1 == o2 || o1 != null && o1.equals(o2);
}
```

Ejercicio 43 (solución propuesta)

```
public static <E> PositionList<E> invertir (PositionList<E> list) {
  if (list == null) return null;
  PositionList<E> res = new NodePositionList<E>();
  invertir(list,list.first(),res);
  return res;
}

// Opcion 1: hacerlo antes de la llamada recursiva
private static <E> void invertir (PositionList<E> list,
                                  Position<E> cursor,
                                  PositionList<E> res) {
  if (cursor != null) {
    res.addFirst(cursor.element());
    invertir(list,list.next(cursor),res);
  }
}

// Opcion 2: hacer primero la llamada recursiva
private static <E> void invertir (PositionList<E> list,
                                  Position<E> cursor,
                                  PositionList<E> res) {
  if (cursor != null) {
    invertir(list,list.next(cursor),res);
    res.addLast(cursor.element());
  }
}
```

Ejercicio 44 (solución propuesta)

```
public static <E> void invertir (PositionList<E> list) {
  if (list == null) throw new IllegalArgumentException ();
  Position<E> cursor = list.first();
  Position<E> cfin = list.last();
  while (cursor != cfin) {
    list.addAfter(cfin, cursor.element());
    Position<E> ctmp = list.next(cursor);
    list.remove(cursor);
    cursor = ctmp;
  }
}
```

Ejercicio 45 (solución propuesta)

```
static <E> PositionList<E> copyElemsNbyN (PositionList<E> list, int n) {
  if (n <= 0) {
    throw new IllegalArgumentException ();
  }
  PositionList<E> listaRes = new NodePositionList<E> ();

  int pos = 0;
  Position<E> cursor = list.first();
  while(cursor != null) {
    if (pos % n == 0) {
      listaRes.addLast(cursor.element());
    }
    pos++;
    cursor = list.next(cursor);
  }

  return listaRes;
}
```

Ejercicio 46 (solución propuesta)

```
public static double average (PositionList<Integer> list) {
  if (list == null || list.isEmpty()) {
    throw new IllegalArgumentException();
  }
  return suma(list, list.first()) / list.size();
}

public static int suma (PositionList<Integer> list,
                        Position<Integer> cursor) {
  if (cursor == null) {
    return 0;
  }
  return cursor.element() + suma(list, list.next(cursor));
}
```

Ejercicio 47 (solución propuesta)

```
public static <E> void quitarIguales(PositionList<E> list, E elem) {
  if (list == null) {
    trow new IllegalArgumentException ("list no puede ser null");
  }

  Position<E> cursor = list.first();
  while (cursor != null) {
    Position<E> nxt = list.next(cursor);
    if (eqNull(cursor.element(),elem)) {
      list.remove(cursor);
    }
    cursor = nxt;
  }
}

private static boolean eqNull(Object o1, Object o2) {
  return o1 == o2 || o1 != null && o1.equals(o2);
}
```

Ejercicio 48 (solución propuesta)

```
public static <E> int countApariciones(PositionList<E> list, E elem) {
  if (list == null) {
    return 0;
  }
  return countApariciones(list,elem,list.first(),0);
}

private static <E> int countApariciones(PositionList<E> list,
                                        E elem,
                                        Position<E> cursor,
                                        int acum) {
  if (cursor == null) {
    return acum;
  }
  acum += (eqNull(cursor.element(),elem)?1:0);
  return countApariciones(list, elem, list.next(cursor),acum);
}
```

Ejercicio 49 (solución propuesta)

```
static void multiplyAndClean (PositionList<Integer> list, int n) {
    Position<Integer> cursor = list.first();
    while (cursor != null) {
        Integer elem = cursor.element();
        Position<Integer> next = list.next(cursor);
        if (elem == null ) {
            list.remove(cursor);
        }
        else {
            list.set(cursor,elem*n);
        }
        cursor = next;
    }
}
```

Ejercicio 50 (solución propuesta)

```
static <E> PositionList<E> copiarHastaElem(PositionList<E> list, E elem) {
  PositionList<E> res = new NodePositionList<E>();

  if (list == null) {
    throw new IllegalArgumentException();
  }

  copiarHastaElemRec(list,elem,list.first(),res);
  return res;

}
static <E> void copiarHastaElemRec(PositionList<E> list,
                            E elem,
                            Position<E> cursor,
                            PositionList<E> res) {
  if (cursor == null || elem.equals(cursor.element())) {
    return;
  }

  res.addLast(cursor.element());
  copiarHastaElemRec(list,elem,list.next(cursor),res);
}
```

- HUECO 1: `if (list == null) throw new IllegalArgumentException();`
- HUECO 2: `list.first(),res`
- HUECO 3: `cursor == null || elem.equals(cursor.element())`
- HUECO 4: `list,elem,list.next(cursor),res`

Ejercicio 185 (solución propuesta)

```
PositionList<Integer> copiarHastaSumN (PositionList<Integer> list,
                                       int n) {
  if (n <= 0) {
    throw new IllegalArgumentException();
  }
  PositionList<Integer> res = new NodePositionList<>();
  Position<Integer> cursor = list.first();
  while (cursor != null && cursor.element() <= n) {
    res.addLast(cursor.element());
    n -= cursor.element();
    cursor = list.next(cursor);
  }

  return res;
}
```

Ejercicio 52 (solución propuesta)

```
public static int sumaElementosPositivos (PositionList<Integer> list) {
  if (list == null) {
    throw new IllegalArgumentException();
  }
  return sumaElementosPositivos(list, list.first());
}

private static int sumaElementosPositivos (PositionList<Integer> list,
                                           Position<Integer> cursor) {
  if (cursor == null) {
    return 0;
  }
  int val = (cursor.element() != null && cursor.element() > 0
             ? cursor.element() : 0);
  return val + sumaElementosPositivos(list, list.next(cursor));
}
```

Ejercicio 53 (solución propuesta)

```
public static <E> PositionList<E> copiarCicular (PositionList<E> list,
                                                 Position<E> pos) {

  PositionList<E> res = new NodePositionList<E>();
  Position<E> cursor = pos;
  int pendientes = list.size();
  while (pendientes > 0) {
    res.addLast(cursor.element());
    cursor = list.next(cursor);
    if (cursor == null) {
      cursor = list.first();
    }
    pendientes --;
  }
  return res;
}
```

Ejercicio 54 (solución propuesta)

```
public static int multiplyAndClean (PositionList<Integer> list, int t) {
  if (list == null) {
    throw new IllegalArgumentException();
  }
  PositionList<Integer> res = new NodePositionList<Integer>();
  multiplyAndClean(list, n, list.first(), res);
  return res;
}

private static void multiplyAndClean (PositionList<Integer> list,
                                      int n,
                                      Position<Integer> cursor,
                                      PositionList<Integer> res) {
  if (cursor == null) {
    return;
  }
  if (cursor.element() != null) {
    res.addLast(cursor.element()*n);
  }
  multiplyAndClean(list, n, list.next(cursor), res);
}
```

Ejercicio 55 (solución propuesta)

Solución:

```
PositionList<Integer> sumar (PositionList<Integer> l1,
                             PositionList<Integer> l2) {

  PositionList<Integer> suma = new NodePositionList<>();

  Position<Integer> c1 = l1.last();
  Position<Integer> c2 = l2.last();

  int acarreo = 0;
  while (c1 != null || c2 != null) {
    int sumadigito = acarreo;
    if (c1 != null) {
      sumadigito += c1.element();
      c1 = l1.prev(c1);
    }
    if (c2 != null) {
      sumadigito += c2.element();
      c2 = l2.prev(c2);
    }
    suma.addFirst(sumadigito % 10);
    acarreo = sumadigito / 10;
  }
  if (acarreo > 0) {
    suma.addFirst(1);
  }
  return suma;
}
```

Ejercicio 56 (solución propuesta)

Solución:

```
  static int cuantosEnRango (PositionList<Integer> list, int a, int b) {
    if (b < a) {
      return 0; ;
    }
    return cuantosEnRango(list, a, b, list.first());
  }

  private static int cuantosEnRango(PositionList<Integer> list,
                                    int a, int b,
                                    Position<Integer> cursor) {
    if (cursor == null) {
      return 0;
    }
    return ((a <= cursor.element() && cursor.element() <= b) ? 1 : 0) +
           cuantosEnRango(list, a, b, list.next(cursor));
  }
```

Ejercicio 57 (solución propuesta)

```
PositionList<Integer> sumaLista (PositionList<PositionList<Integer>> lista) {

  PositionList<Integer> res = new NodePositionList<>();
  Position<PositionList<Integer>> cout = lista.first();

  while (cout != null) {
    res.addLast (suma(cout.element()));
    cout = lista.next(cout);
  }

  return res;
}

int suma (PositionList<Integer> l) {
  Position<Integer> c = l.first();
  int suma = 0;
  while (c != null) {
    suma += c.element();
    c = l.next(c);
  }
  return suma;
}
```

Ejercicio 58 (solución propuesta)

```
static boolean hayEnRango (PositionList<Integer> list, int a, int b) {
  if (b < a) {
    return false;
  }
  return hayEnRango(list, a, b, list.first());
}

private static boolean hayEnRango(PositionList<Integer> list,
                                  int a, int b,
                                  Position<Integer> cursor) {
  if (cursor == null) {
    return false;
  }
  if (a <= cursor.element() && cursor.element() <= b) {
    return true;
  }
  return hayEnRango(list, a, b, list.next(cursor));
}
```

Ejercicio 59 (solución propuesta)

```
public static <E> PositionList<E> elementosUnicos (PositionList<E> lista) {
  PositionList<E> res = new NodePositionList<>();

  Position<E> cursor = lista.first();
  while (cursor != null) {
    if (!estaRepetido(lista,cursor.element())) {
      res.addLast(cursor.element());
    }
    cursor = lista.next(cursor);
  }
  return res;
}

private static <E> boolean estaRepetido (PositionList<E> list, E element) {
  Position<E> cursor = list.first();
  int count = 0;
  while (cursor != null && count < 2) {
    if (eqNull(element, cursor.element())) {
      count ++;
    }
    cursor = list.next(cursor);
  }
  return count > 1;
}

private static boolean eqNull(Object o1, Object o2) {
  return o1 == o2 || o1 != null && o1.equals(o2);
}
```

Ejercicio 60 (solución propuesta)

```
public static <E> PositionList<E> barajar (PositionList<E> lista) {
  PositionList<E> res = new NodePositionList<>();
  if (lista.size() % 2 == 0) {
    throw new IllegalArgumentException();
  }
  barajar(lista,lista.first(),lista.last(),res);
  return res;
}

private static <E> void barajar(PositionList<E> lista, Position<E> ini,
                                Position<E> fin, PositionList<E> res) {
  if (ini == fin) {
    res.addLast(ini.element()); // o res.addLast(fin.element());
    return;
  }
  res.addLast(ini.element());
  res.addLast(fin.element());
  barajar(lista,lista.next(ini),lista.prev(fin),res);
}
```

Ejercicio 61 (solución propuesta)

```
public static <E> PositionList<E> elementosRepetidos (PositionList<E> lista) {
  PositionList<E> res = new NodePositionList<>();

  Position<E> cursor = lista.first();
  while (cursor != null) {
    if (estaRepetidoDesde(lista,cursor.element(),lista.next(cursor)) &&
        !member(res,cursor.element())) {
      res.addLast(cursor.element());
    }
    cursor = lista.next(cursor);
  }
  return res;
}

private static <E> boolean estaRepetidoDesde (PositionList<E> list,
                                              E element,
                                              Position<E> cursor) {

  while (cursor != null && !eqNull(element, cursor.element())) {
    cursor = list.next(cursor);
  }
  return cursor != null;
}
```

Otra posible solución un poco menos eficiente:

```
public static <E> PositionList<E> elementosRepetidos (PositionList<E> lista) {
  PositionList<E> res = new NodePositionList<>();

  Position<E> cursor = lista.first();
  while (cursor != null) {
    if (!member(res,cursor.element()) &&
        estaRepetido(lista,cursor.element())) {
      res.addLast(cursor.element());
    }
    cursor = lista.next(cursor);
  }
  return res;
}

private static <E> boolean estaRepetido (PositionList<E> list, E element) {
  Position<E> cursor = list.first();
  int count = 0;
  while (cursor != null && count < 2) {
    if (eqNull(element, cursor.element())) {
      count ++;
    }
    cursor = list.next(cursor);
  }
  return count > 1;
}

private static boolean eqNull(Object o1, Object o2) {
  return o1 == o2 || o1 != null && o1.equals(o2);
```

Ejercicio 62 (solución propuesta)

```
public static <E> PositionList<E> intercalar (PositionList<E> l1,
                                              PositionList<E> l2) {
  if (l1.size() != l2.size()) {
    throw new IllegalArgumentException();
  }
  PositionList<E> res = new NodePositionList<>();
  intercalar(l1,l2,l1.first(),l2.first(),res);
  return res;
}

private static <E> void intercalar(PositionList<E> l1,
                                   PositionList<E> l2,
                                   Position<E> c1,
                                   Position<E> c2,
                                   PositionList<E> res) {
  if (c1 == null) {
    return;
  }
  res.addLast(c1.element());
  res.addLast(c2.element());
  intercalar(l1, l2, l1.next(c1), l2.next(c2),res);
}
```

Ejercicio 63 (solución propuesta)

```
public static <E> PositionList<E> expandir (PositionList<Pair<E,Integer>> lista) {
  PositionList<E> resultado = new NodePositionList<>();

  if (lista == null) {
    throw new IllegalArgumentException();
  }

  Position<Pair<E,Integer>> cursor = lista.first();

  while (cursor != null) {

    E elemento = cursor.element().getLeft();
    Integer apariciones = cursor.element().getRight();
      for (int i = 0; i < apariciones; i++) {
        resultado.addLast(elemento);
      }
    cursor = lista.next(cursor);
  }

  return resultado;
}
```

Ejercicio 64 (solución propuesta)

```
static <E> PositionList<E> expandirRec (PositionList<Pair<E,Integer>> lista) {
  if (lista == null) {
    throw new IllegalArgumentException();
  }

  PositionList<E> res = new NodePositionList<>();

  expandirRec(lista,lista.first(),res);
  return res;
}

private static <E> void expandirRec(PositionList<Pair<E, Integer>> lista,
                                    Position<Pair<E, Integer>> cursor,
                                    PositionList<E> res) {
  if (cursor == null) {
    return;
  }
  clonar(cursor.element().getLeft(), cursor.element().getRight(), res);

  expandirRec(lista,lista.next(cursor),res);
}

private static <E> void clonar (E e, Integer i, PositionList<E> res) {
  if (i > 0) {
    res.addLast(e);
    clonar(e,i-1,res);
  }
}
```

Ejercicio 66 (solución propuesta)

```
public void addBefore(Position<E> p, E e) {
  Node<E> newNode = new Node<E>(p.getPrev(), p, e);
  numElts++;
  p.getPrev().setNext(newNode);
  p.setPrev(newNode);
}
```

Ejercicio 67 (solución propuesta)

(d)

Ejercicio 68 (solución propuesta)

```
void append(NodePositionList<E> list) {
  if (!list.isEmpty()) {
    if (isEmpty()) {
      /* swap headers and trailers */
      Node<E> tmpHeader, tmpTrailer;
      tmpHeader    = header;
      tmpTrailer   = trailer;
      header       = list.header;
      trailer      = list.trailer;
      list.header  = tmpHeader;
      list.trailer = tmpTrailer;
    } else {
      /* both lists not empty */
      trailer.getPrev().setNext(list.header.getNext());
      trailer.getPrev().getNext().setPrev(trailer.getPrev());
      Node<E> oldThisTrailer = trailer;  /* saved for reuse */
      trailer = list.trailer;
      /* now empty the argument list */
      list.trailer = oldThisTrailer;
      list.trailer.setPrev(list.header);
      list.header.setNext(list.trailer);
    }
    numElts += list.numElts;
    list.numElts = 0;
  }
}
```

Ejercicio 69 (solución propuesta)

```
  void swap(NodePositionList<E> list) {
    if (list != this) {
      Node<E> tmpHeader, tmpTrailer;
      int tmpNumElts;
      tmpHeader    = header;
      tmpTrailer   = trailer;
      tmpNumElts   = numElts;
      header       = list.header;
      trailer      = list.trailer;
      list.header  = tmpHeader;
      list.trailer = tmpTrailer;
      numElts      = list.numElts;
      list.numElts = tmpNumElts;
    }
  }
```

Ejercicio 70 (solución propuesta)

```
p.getNext().getNext().setPrev(p);
p.setNext(p.getNext().getNext());
```

Ejercicio 71 (solución propuesta)

```
q.setNext(p.getNext());
q.setPrev(p);
p.getNext().setPrev(q);
p.setNext(q);
```

Ejercicio 72 (solución propuesta)

(c)

Ejercicio 73 (solución propuesta)

```
p.getPrev().setNext(p.getNext()) ;
p.getNext().setPrev(p.getPrev()) ;
p.setPrev(null) ;
p.setNext(null) ;
```

Ejercicio 74 (solución propuesta)

1. El constructor debe recibir `header.getNext()` en vez de `trailer` pues la lista podría no estar vacía.
2. No se incrementa el atributo `numElts` que lleva el número de elementos en la lista.

Ejercicio 75 (solución propuesta)

```
nuevo.setNext(n.getNext());
nuevo.setPrev(n);
n.getNext().setPrev(nuevo);
n.setNext(nuevo);
```

Ejercicio 76 (solución propuesta)

1) Solución 1 (usando **for**-each):

```
public static<E> FIFO<E> reverse (FIFO<E> q) {
  LIFO<E> s = new LIFOList<E>();
  for (E e : q)
    s.push(e);
  FIFO<E> r = new FIFOList<E>();
  for (E e : s)
    r.enqueue(e);
  return r;
}
```

2) Usando los constructores de `FIFOList` y `LIFOList`

```
public static<E> FIFO<E> reverse (FIFO<E> q) {
  return new
     FIFOList<E>((new LIFOList<E>(q.toPositionList())).toPositionList());
}
```

3) Otros bucles (hay más opciones correctas que las presentadas en esta solución)

```
public static<E> FIFO<E> reverse (FIFO<E> q) {
  FIFO<E> resultado = new FIFOList<E>();
  LIFO<E> pila = new LIFOList<E> ();

  Iterator<E> it = q.iterator();
  while(it.hasNext())
    pila.push(it.next());

  while(!pila.isEmpty()) {
    resultado.enqueue(pila.top());
    pila.pop();
  }

  return resultado;
}
```

Ejercicio 77 (solución propuesta)

```
public static boolean estaEquilibrado (Character [] texto) {
  boolean error = false;
  LIFO<Character> lifo = new LIFOList<Character> ();
  for (int i = 0; i < texto.length && !error; i ++) {
    switch (texto[i]) {
    case '(':
    case '{':
      lifo.push(texto[i]);
      break;
    case ')':
    case '}':
      if (lifo.isEmpty()) {
        error = true;
      }
      else if ((texto[i].equals('}') && !lifo.top().equals('{')) ||
               (texto[i].equals(')') && !lifo.top().equals('(')) ) {
        error = true;
      }
      else {
        lifo.pop();
      }
      break;
    }
  }
  return !error && lifo.isEmpty();
}
```

Ejercicio 78 (solución propuesta)

Solución:

```
public class LIFOList<E> implements LIFO<E> {
  private PositionList<E> elementos;

  public LIFOList () {
    elementos = new NodePositionList<E>();
  }

  public void push (E e) {
    elementos.addFirst(e);
  }

  public E top () throws EmptyStackException {
    if (elementos.size()==0) throw new EmptyStackException ();
    return elementos.first().element();
  }

  public E pop () throws EmptyStackException {
    if (elementos.size()==0) throw new EmptyStackException ();
    return elementos.remove(elementos.first());
  }

  public int size () {
    return elementos.size();
  }

  public int isEmpty () {
    return elementos.size() == 0;
  }

}
```

Ejercicio 79 (solución propuesta)

```
static class ControlNavegador {
  private String actual = null;
  private LIFO<String> pilaAtras = new LIFOList<String>();

  public void nuevaURL (String url) {
    if (actual != null) {
      pilaAtras.push(actual);
    }
    actual = url;
  }

  public String atras () throws AtrasNoPosibleException{
    if (pilaAtras.isEmpty()) throw new AtrasNoPosibleException();
    actual = pilaAtras.pop();
    return actual;
  }
}
```

Ejercicio 80 (solución propuesta)

```
public class FIFOList<E> implements FIFO<E> {
  private PositionList<E> elementos;

  public FIFOList () {
    elementos = new NodePositionList<E>();
  }

  public void enqueue (E e) {
    elementos.addFirst(e);
  }

  public E first () throws EmptyFIFOException {
    if (elementos.size()==0) throw new EmptyFIFOException ();
    return elementos.last().element();
  }

  public E dequeue () throws EmptyFIFOException {
    if (elementos.size()==0) throw new EmptyFIFOException ();
    return elementos.remove(elementos.last());
  }

  public int size () {
    return elementos.size();
  }

  public int isEmpty () {
    return elementos.size() == 0;
  }

}
```

Ejercicio 81 (solución propuesta)

```
  private void moveCursor () {
    while (cursor != null &&
           (cursor.element()== null || cursor.element() <= 0)) {
       cursor = list.next(cursor);
    }
}
```

Ejercicio 82 (solución propuesta)

```
public NonNullElementIterator(PositionList<E> list) {
  if (list == null) throw new NullPointerException();
  else {
    this.it         = list.iterator();
    this.cursorElem = getNonNull();
  }
}
public E next() throws NoSuchElementException {
  if (this.cursorElem == null) throw new NoSuchElementException();
  else {
    E toReturn      = this.cursorElem;
    this.cursorElem = getNonNull();
    return toReturn;
  }
}
/* Returns non-null element or null if no such element */
private E getNonNull() {
  E e = null;
  while (this.it.hasNext() && (e = this.it.next()) == null)
    ;
  return e;
}
```

Ejercicio 83 (solución propuesta)

```
private Position<E> skipElem(Position<E> cur) {
  while (cur != null && cur.element().equals(this.elem))
    cur = this.list.next(cur);
  return cur;
}
public SkipIterator(PositionList<E> list, E elem) {
  this.list   = list;
  this.elem   = elem;
  this.cursor = skipElem(this.list.first());
}
public E next() throws NoSuchElementException {
  if (this.cursor == null) throw new NoSuchElementException();
  else {
    E toReturn  = this.cursor.element();
    this.cursor = skipElem(this.list.next(this.cursor));
    return toReturn;
  }
}
```

Ejercicio 84 (solución propuesta)

```
static Integer minimo (Iterable<Integer> iterable) {
  Integer minimo = -1;
  Iterator<Integer> it = iterable.iterator();
  while (it.hasNext()) {
    Integer actual = it.next();
    if (actual != null && (minimo == -1 || actual < minimo)) {
      minimo = actual;
    }
  }
  return
minimo;
}
```

Ejercicio 85 (solución propuesta)

```
public static <E> int findPos (Iterable<E> iterable, E elem) {
  int pos = 0;
  int res = -1;
  Iterator<E> it = iterable.iterator();

  while (it.hasNext() && res == -1) {
    if (it.next().equals(elem)) {
      res = pos;
    }
    pos ++;
  }
  return res;
}
```

Ejercicio 86 (solución propuesta)

```
static <E> void imprimirSinConsecutivosIguales (Iterable<E> iterable) {
  Iterator<E> it = iterable.iterator();
  E last = null;
  if (it.hasNext()) {
    last = it.next();
    System.out.println(last);
  }
  while (it.hasNext()) {
    E actual = it.next();
    if (!eqNull(last,actual)) {
      System.out.println(actual);
    }
    last = actual;
  }
}

private static boolean eqNull(Object o1, Object o2) {
  return o1 == o2 || o1 != null && o1.equals(o2);
}
```

Ejercicio 87 (solución propuesta)

```
public static <E> boolean iguales (Iterable<E> iterable1,
                                   Iterable<E> iterable2) {
  Iterator<E> it1 = iterable1.iterator();
  Iterator<E> it2 = iterable2.iterator();

  boolean iguales = true;
  while (it1.hasNext() && it2.hasNext() && iguales) {
    iguales = eqNull (it1.next(), it2.next());
  }
  return iguales && (it1.hasNext() == it2.hasNext()) ;
}

public static boolean eqNull (Object o1, Object o2) {
  return o1 == o2 || (o1!= null && o1.equals(o2));
}
```

Ejercicio 88 (solución propuesta)

```
static <E> boolean areAllGreaterThan (Iterable<Integer> iter, Integer elem) {
  Iterator<Integer> it = iter.iterator();
  boolean allGreater = true;
  while (it.hasNext() && allGreater) {
    Integer val = it.next();
    if (val != null) {
      allGreater = val > elem;
    }
  }
  return allGreater;
}
```

Ejercicio 89 (solución propuesta)

```
public static Integer primerDesordenado (Iterable<Integer> iterable) {
  Iterator<E> it = iterable.iterator();

  if (!it.hasNext()) {
    return null;
  }
  E result = null;
  E last = it.next();

  while (it.hasNext() && result == null) {
    E elem = it.next();
    if (last.compareTo(elem) > 0) { // o (last > elem)
      result = elem;
    }
    last = elem;
  }
  return result;
}
```

Ejercicio 90 (solución propuesta)

Solución:

```
class IteradorCircular<E> implements Iterator<E> {
  private IndexedList<E> list;
  private int pos;
  private int fin;
  private boolean started;

  public IteradorCircular(IndexedList<E> list, int initPos) {
    if (n >= list.size()|| n < 0) {
      throw new IllegalArgumentException ();
    }
    this.list = list;
    this.pos = initPos;
    this.fin = initPos;
    started = false;
  }

  public boolean hasNext() {
    return fin != pos || !started;
  }

  public E next() {
    if (!hasNext()) {
      throw new NoSuchElementException();
    }
    E v = list.get(pos);
    pos = (pos + 1) % list.size();
    started = true;
    return v;
  }
}
```

Ejercicio 91 (solución propuesta)

```
static Integer maximo (Iterable<Integer> iterable) {
  Iterator<Integer> it = iterable.iterator();
  if (!it.hasNext()) {
    throw new IllegalArgumentException();
  }
  Integer maximo = it.next();
  while (it.hasNext()) {
    Integer actual = it.next();
    if (actual > maximo) {
      maximo = actual;
    }
  }
  return maximo;
}
```

Ejercicio 92 (solución propuesta)

Solución:

```
static <E> PositionList<E> recolectardeNenN (Iterable<E> iterable, int n) {
  if (n <= 0) {
    throw new IllegalArgumentException();
  }
  PositionList<E> res = new NodePositionList<>();

  Iterator<E> it = iterable.iterator();
  int pos = 0;
  while (it.hasNext()) {
    E e = it.next();
    if (pos % n == 0) {
      res.addLast(e);
    }
    pos ++;
  }
  return res;
}
```

Ejercicio 93 (solución propuesta)

```
static PositionList<Integer> getDistancias (Iterable<Integer> iterable) {
  PositionList<Integer> res = new NodePositionList<>();

  Iterator<Integer> it = iterable.iterator();

  if (!it.hasNext()) {
    throw new IllegalArgumentException();
  }

  Integer prev = it.next();

  if (!it.hasNext()) {
    throw new IllegalArgumentException();
  }

  while (it.hasNext()) {
    Integer elem = it.next();
    res.addLast(Math.abs(prev-elem));
    prev = elem;
  }

  return res;
}
```

Ejercicio 94 (solución propuesta)

```
  public static boolean esSerieGeometrica (Iterable<Integer> iterable) {
    Iterator<Integer> it = iterable.iterator();
    boolean esSerie = true;

    Integer prev = null;
    if (it.hasNext()) {
      prev = it.next();
    }

    while (it.hasNext() && esSerie) {
      Integer current = it.next();
      esSerie = current.equals(prev * 2);
      prev = current;
    }
    return esSerie;
  }
```

Ejercicio 95 (solución propuesta)

```
public static PositionList<Pair<Integer,Integer>> getDistanciasMayores
                                        (Iterable<Integer> iterable,
                                         int min) {

 if (min < 0) {
   throw new IllegalArgumentException();
 }

 Iterator<Integer> it = iterable.iterator();

 PositionList<Pair<Integer,Integer>> res = new NodePositionList<>();

 Integer prev = null;
 if (it.hasNext()) {
   prev = it.next();
 }

 while (it.hasNext()) {
   Integer current = it.next();
   int distancia = Math.abs(current-prev);
   if (distancia > min) {
     res.addLast(new Pair<>(prev,current));
   }
   prev = current;
 }
 return res;
}
```

Ejercicio 96 (solución propuesta)

```
  static <E> PositionList<Pair<E,Integer>> compactar (Iterable<E> iterable) {
    PositionList<Pair<E,Integer>> resultado = new NodePositionList<>();

    Iterator<E> it = iterable.iterator();
    if (!it.hasNext()) {
      return resultado;
    }
    resultado.addLast(new Pair<E,Integer>(it.next(),1));

    while (it.hasNext()) {
      E element = it.next();
      Pair<E,Integer> last = resultado.last().element();

      if (eqNull(element,last.getLeft())) {
        last.setRight(last.getRight()+1);
      }
      else {
        resultado.addLast(new Pair<E,Integer>(element,1));
      }
    }

    return resultado;

  }

  private static boolean eqNull(Object o1, Object o2) {
    return o1 == o2 || o1 != null && o1.equals(o2);
  }
}
```

Ejercicio 98 (solución propuesta)

Preorden: [1, 2, 5, 9, 6, 10, 11, 3, 4, 7, 8]
Postorden: [9, 5, 11, 10, 6, 2, 3, 7, 8, 4, 1]

Ejercicio 99 (solución propuesta)

```
public static <E> PositionList<E> getExternalChildren(Tree<E> t) {
  PositionList<E> res = new NodePositionList<E>();
  if (t.isEmpty()) {
    return res;
  }
  getExternalChildren(t,t.root(),res);
  return res;
}

public static <E> void getExternalChildren(Tree<E> t,
                                           Position<E> node,
                                           PositionList<E> res) {

  boolean found = false;
  Iterator<Position<E>> it = t.children(node).iterator();
  while (it.hasNext() && !found) {
    found = t.isExternal(it.next());
  }

  if (found) {
    res.addLast(node.element());
  }

  for (Position<E> w: t.children(node)) {
    getExternalChildren(t,w,res);
  }
}
```

Ejercicio 100 (solución propuesta)

Preorden: [1, 2, 5, 6, 11, 7, 3, 8, 4, 9, 10, 12, 13]
Postorden: [5, 11, 6, 7, 2, 8, 3, 9, 13, 12, 10, 4, 1]

Ejercicio 101 (solución propuesta)

```
public static <E> PositionList<E> leaves(Tree<E> t) {
  PositionList<E> res = new NodePositionList<E>();
  if (t.isEmpty()) {
    return res;
  }
  leaves(t,t.root(),res);
  return res;
}

public static <E> void leaves(Tree<E> t, Position<E> node, PositionList<E> res)
{

  if (t.isExternal(node)) {
    res.addLast(node.element());
  }
  for (Position<E> w: t.children(node)) {
    leaves(t,w,res);
  }
}
```

Ejercicio 102 (solución propuesta)

```
public boolean siblings(Tree<E> t, Position<E> p1, Position<E> p2) {
  return p1 != p2  && !t.isRoot(p1) && !t.isRoot(p2)
         && t.parent(p1) == t.parent(p2) ;
}
```

Ejercicio 103 (solución propuesta)

```
public static <E> PositionList<E> ancestros(Tree<E> t, Position<E> p) {
  PositionList<E> r = new NodePositionList<E>();
  while (!t.isRoot(p)) {
    p = t.parent(p);
    r.addLast(p.element());
  }
  return r;
}
```

Ejercicio 104 (solución propuesta)

```
static <E> void imprimirCaminosHojas(Tree<E> tree) {
  if (tree.isEmpty()) return;

  imprimirTodosCaminos(tree,tree.root(),"");
}

static <E> void imprimirCaminosHojas(Tree<E> tree,
                                     Position<E> v, String path) {
  path += v.element().toString();

  if (tree.isExternal(v))
    System.out.println(path);

  for (Position<E> w : tree.children(v))
    imprimirCaminosHojas(tree, w, path);
}
```

Ejercicio 105 (solución propuesta)

```
public static boolean hasHeapPropertyGen(Tree<Integer> tree) {
    return tree.isEmpty() || hasHeapPropertyGen(tree, t.root());
}

public static boolean hasHeapPropertyGen (Tree<Integer> tree,
                                          Position<Integer> node) {

  if (tree.parent(node) != null && node.element() < tree.parent(node).element()) {
    return false;
  }

  Iterator<Position<Integer>> it = tree.children(node).iterator();
  boolean res = true;
  while(it.hasNext() && res) {
    res = hasHeapPropertyGen(tree,it.next());
  }
  return res;
}
```

Ejercicio 106 (solución propuesta)

```
public static double maximoCamino (BinaryTree<Integer> tree) {
  if (tree == null) {
    throw new IllegalArgumentException();
  }
  return maximoCamino(tree, tree.root());
}

public static Integer maximoCamino (BinaryTree<Integer> tree,
                                    Position<Integer> node) {

  Integer sumLeft = 0;
  Integer sumRight = 0;

  if (tree.hasLeft(node)) {
    sumLeft = maximoCamino(tree,tree.left(node));
  }
  if (tree.hasRight(node)) {
    sumRight = maximoCamino(tree,tree.right(node));
  }

  return node.element() + Math.max(sumLeft,sumRight);
}

/// Otra posible solucion

public static double maximoCamino (BinaryTree<Integer> tree) {
  if (tree == null) {
    throw new IllegalArgumentException();
  }
  return maximoCamino(tree, tree.root(), 0, 0);
}

public static Integer maximoCamino (BinaryTree<Integer> tree,
                                    Position<Integer> node,
                                    int current,
                                    int maximo) {

  current += node.element();

  if (tree.isExternal(node)) {
    return Math.max(maximo, current);
  }

  if (tree.hasLeft(node)) {
    maximo = maximoCamino(tree,tree.left(node), current, maximo);
  }
  if (tree.hasRight(node)) {
    maximo = maximoCamino(tree,tree.right(node), current, maximo);
  }
  return maximo;
}
```

Ejercicio 107 (solución propuesta)

```
sttic <E> boolean member (Tree<E> tree, E elem) {
  if (tree.isEmpty()) {
    return false;
  }
  return member (tree, elem, tree.root());
}

static <E> boolean member (Tree<E> tree,
                           E elem,
                           Position<E> node) {

  if (node.element().equals(elem)) {
    return true;
  }
  if (tree.isExternal(node)) {
    return false;
  }
  Iterator<Position<E>> it = tree.children(node).iterator();
  boolean found = false;
  while (it.hasNext() && !found) {
    found = member(tree,elem,it.next());
  }
  return found;
}
```

Ejercicio 108 (solución propuesta)

```
  static <E> boolean existeHoja (Tree<E> tree, E e) {
    return existeHoja(tree,e,tree.root());
  }

  private static <E> boolean existeHoja(Tree<E> tree,
                                        E e,
                                        Position<E> cursor) {
    if (cursor == null) {
      return false;
    }
    if (tree.isExternal(cursor) && eqNull(e, cursor.element())) {
      return true;
    }

    Iterator<Position<E>> it = tree.children(cursor).iterator();
    boolean encontrado = false;
    while (it.hasNext() && !encontrado) {
      encontrado = existeHoja(tree,e,it.next());
    }

    return encontrado;
  }

  private static boolean eqNull(Object o1, Object o2) {
    return o1 == o2 || o1 != null && o1.equals(o2);
  }
```

Ejercicio 109 (solución propuesta)

```
public static <E> E leftmostLeaf(BinaryTree<E> t) throws EmptyTreeException {
    if (t.isEmpty()) throw new EmptyTreeException("El árbol está vacío");
    Position <E> pos=t.root();
    while (!t.isExternal(pos)) {
        pos = t.hasLeft(pos) ? t.left(pos) : t.right(pos);
    }
    return pos.element();
}
```

Ejercicio 110 (solución propuesta)

(a) No, es imposible. El mínimo es $O(n)$, pues en el peor caso compararíamos n elementos (todas las comparaciones darían la igualdad como resultado).

(b)
```
public class BinaryTreeComparator<E> implements Comparator<BinaryTree<E>> {

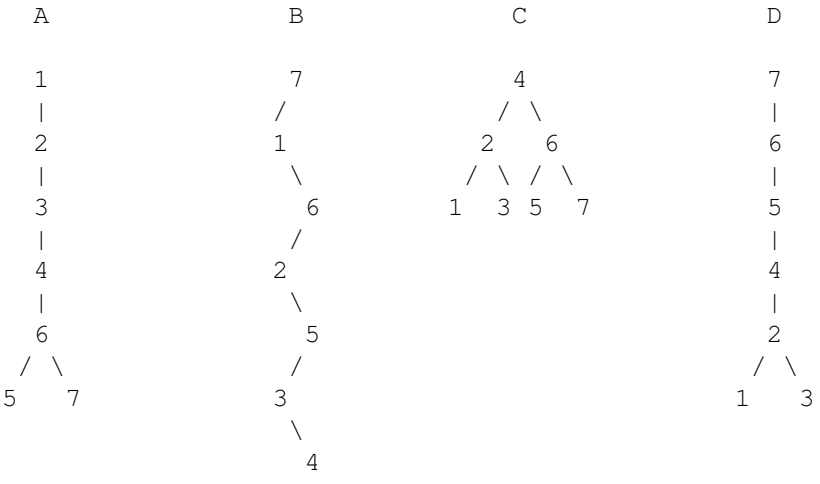
   private Comparator<E> compElementos;

   public BinaryTreeComparator(Comparator<E> compElementos) {
      this.compElementos = compElementos;
   }

   public int compare(BinaryTree<E> t1, BinaryTree<E> t2) {
      if (t1.isEmpty() && t2.isEmpty()) return 0;
      Iterator<E> it1 = t1.iterator();
      Iterator<E> it2 = t2.iterator();
      int resultado = 0;
      while ( it1.hasNext() &&
              it2.hasNext() &&
              (resultado = compElementos.compare(it1.next(),it2.next())) == 0 )
        ;
      if (resultado != 0)
         return resultado;
         else return t1.size() - t2.size();
     }
}
```

Ejercicio 111 (solución propuesta)

(b)
Estos son los arboles resultantes:



Ejercicio 113 (solución propuesta)

La altura máxima es $n-1$, que se da si todos los nodos internos del árbol tienen únicamente un hijo. La altura mínima es $\log n$, que se da si todos los nodos internos del árbol tienen dos hijos, exceptuando los del penúltimo nivel que pueden tener sólo un hijo.

Ejercicio 114 (solución propuesta)

```
BinaryTree<String> t = new LinkedBinaryTree<String>();
Position<String> p;
p = t.addRoot("un");
t.insertLeft(p, "en");
t.insertRight(p, "de");
p = t.right(p);
t.insertLeft(p, "lugar");
t.insertRight(p, "la");
p = t.right(p);
t.insertRight(p, "Mancha");
```

Ejercicio 115 (solución propuesta)

(a) un en de lugar la Mancha

(b) en un lugar de la Mancha

(c) en lugar Mancha la de un

Ejercicio 116 (solución propuesta)

(a) Es un recorrido postorden.

(b) Es un recorrido arbitrario.

Ejercicio 117 (solución propuesta)

```
LinkedBinaryTree<Integer> tree = new LinkedBinaryTree<Integer>();
tree.addRoot(3);
tree.insertLeft(tree.root(), 6);
tree.insertRight(tree.root(), 7);

tree.insertLeft(tree.left(tree.root()), 11);
tree.insertRight(tree.left(tree.root()), 12);

tree.insertLeft(tree.right(tree.root()), 13);
tree.insertRight(tree.right(tree.root()), 14);

tree.insertRight(tree.left(tree.right(tree.root())), 18);
```

Existen otras secuencias válidas posibles según qué métodos del interfaz `BinaryTree<E>` se utilicen.

Ejercicio 118 (solución propuesta)

```
  public void inOrder(BinaryTree<E> t) {
    if (!t.isEmpty()) inOrderAux(t,t.root());
  }

  public void inOrderAux(BinaryTree<E> t, Position<E> p) {
      if (t.hasLeft(t,p))  inOrderAux(t, t.left(p));
      System.out.print(p.element().toString() + " ");
      if (t.hasRight(t,p)) inOrderAux(t, t.right(p));
  }
```

Ejercicio 119 (solución propuesta)

```
public NodePositionList<E> breadth(BinaryTree<E> tree) {
 NodePositionList<E> res  = new NodePositionList<E>();
 if (!tree.isEmpty()) {
   NodePositionList<Position<E>> cola =
          new NodePositionList<Position<E>>();
   cola.addLast(tree.root());
   while (!cola.isEmpty()) {
     Position<E> p = cola.first().element();
     cola.remove(cola.first());
     res.addLast(p.element());
     for (Position<E> child : tree.children(p)) {
       cola.addLast(child);
     }
  }
 }
 return res;
}
```

Ejercicio 120 (solución propuesta)

1. La complejidad es $O(h)$ donde h es la altura del árbol. En caso peor, la espina es el camino de mayor altura.

2. ```
   public void leftSpine(BinaryTree<E> tree) {
       if (!tree.isEmpty()) {
         Position<E> p = tree.root();
         do {
           tree.removeSubTree(tree.right(p));.
           p = tree.left(p);
         } while (tree.isExtrenal(p)) ;
       }
     }
   ```
   ```

Ejercicio 121 (solución propuesta)

```
public static <E> BinaryTree<E>espejo (BinaryTree<E> t) {
  BinaryTree<E> res = new LinkedBinaryTree<>();
  res.addRoot(t.root().element());
  espejo(t,res,t.root(),res.root());
  return res;
}

public static <E> void espejo(BinaryTree<E> t, BinaryTree<E> res,
                              Position<E> p, Position<E> pres) {

  if (t.hasRight(p)) {
    res.insertLeft(pres, t.right(p).element());
    espejo(t,res,t.right(p),res.left(pres));
  }

  if (t.hasLeft(p)) {
    res.insertRight(pres, t.left(p).element());
    espejo(t,res,t.left(p),res.right(pres));
  }
}
```

Ejercicio 122 (solución propuesta)

(c)

Ejercicio 123 (solución propuesta)

```
public Position<E> leftSibling(BinaryTree<E> t, Position<E> p)
  throws BoundaryViolationException {
  if (!t.isRoot(p) && t.hasLeft(t.parent(p))
                   && t.left(t.parent(p)) != p)
    return t.left(t.parent(p));
  else throw new BoundaryViolationException();
}
```

Ejercicio 124 (solución propuesta)

```
public Position<E> sibling(BinaryTree<E> tree, Position<E> p) {
  if (tree.isRoot(p)) return null;
  else {
    Position<E> sibling = null;
    if (tree.hasLeft(tree.parent(p))
        && tree.left(tree.parent(p)) == p
        && tree.hasRight(tree.parent(p)))
      sibling = tree.right(tree.parent(p));
    else if (tree.hasRight(tree.parent(p))
             && tree.right(tree.parent(p)) == p
             && tree.hasLeft(tree.parent(p)))
      sibling = tree.left(tree.parent(p));

    return sibling;
  }
}
```

Ejercicio 125 (solución propuesta)

El método devuelve la hoja más a la derecha alcanzable desde la raíz.

Ejercicio 126 (solución propuesta)

```
public Position<E> leftSibling(BinaryTree<E> tree, Position<E> p)
{
  if (tree.isRoot(p)) return null;
  else {
    Position<E> sibling = null;
    if (tree.hasRight(tree.parent(p))   &&
        tree.right(tree.parent(p)) == p &&
        tree.hasLeft(tree.parent(p)))
      sibling = tree.left(tree.parent(p));

    return sibling;
  }
}
```

Ejercicio 127 (solución propuesta)

```
  public boolean hasSibling(BinaryTree<E> tree, Position<E> p) {
    return !tree.isRoot(p)
           && tree.hasLeft(tree.parent(p))
           && tree.hasRight(tree.parent(p));
  }
```

Ejercicio 128 (solución propuesta)

```
public static void sumTree(BinaryTree<Integer> tree) {
  if (tree != null && ! tree.isEmpty())
    sumTreeRec(tree.root(),tree);
}

private static void sumTreeRec(Position<Integer> p,
                               BinaryTree<Integer> tree) {
  if (!tree.isExternal(p)) {
    Integer sl = 0, sr = 0;
    if (tree.hasLeft(p)) {
      sumTreeRec(tree.left(p),tree);
      sl = tree.left(p).element();
    }
    if (tree.hasRight(p)) {
      sumTreeRec(tree.right(p),tree);
      sr = tree.right(p).element();
    }
    tree.replace(p, sl + sr);
  }
}
```

Solución alternativa:

```
 private static int sumTreeRec(Position<Integer> p,
                               BinaryTree<Integer> tree) {
  if (tree.isExternal(p)) return p.element();
  else {
    Integer sl = 0, sr = 0;
    if (tree.hasLeft(p))
      sl = sumTreeRec(tree.left(p),tree);
    if (tree.hasRight(p))
      sr = sumTreeRec(tree.right(p),tree);
    tree.replace(p, sl + sr);
    return sl + sr;
  }
}
```

Ejercicio 129 (solución propuesta)

```
public static <E> Position<E> hojaMasIzquierda(BinaryTree<E> tree) {
  if (tree == null || tree.isEmpty()) return null;
  return hmiRec(tree,tree.root());
}

private static <E> Position<E> hmiRec(BinaryTree<E> tree, Position<E> p) {
  if (tree.hasLeft(p))
    return hmiRec(tree,tree.left(p));
  else if (tree.hasRight(p))
    return hmiRec(tree,tree.right(p));
  else
    return p;
}
```

Variante de `hmiRec` usando `? :` en lugar de **`if`**-then-**`else`**:

```
private static <E> Position<E> hmiRec(BinaryTree<E> tree, Position<E> p) {
  return tree.hasLeft(p) ? hmiRec(tree,tree.left(p)) :
         (tree.hasRight(p) ? hmiRec(tree,tree.right(p)) : p);
}
```

Ejercicio 130 (solución propuesta)

Con el `hasLeft` y `hasRight`:

```
public static <E> int ocurrencias(BinaryTree<E> tree, E elem) {
  if (tree.isEmpty()) return 0;
  return ocurrenciasRec(tree, tree.root(), elem);
}

public static <E> int ocurrenciasRec(BinaryTree<E> tree,
                                     Position<E> node, E elem) {
  int ocurrencias = 0;
  if (eqNull(elem,node.element()))
    ocurrencias ++;
  if (tree.hasLeft(node))
    ocurrencias += ocurrenciasRec(tree, tree.left(node), elem);
  if (tree.hasRight(node))
    ocurrencias += ocurrenciasRec(tree, tree.right(node), elem);
  return ocurrencias;
}

public static <E> boolean eqNull (E o1, E o2) {
  return o1==o2 || o1!=null && o1.equals(o2);
}
```

Con `children`

```
public static <E> int ocurrenciasRec(BinaryTree<E> tree,
                       Position<E> node,
                       E elem) {
  int ocurrencias = 0;
  if (eqNull(elem,node.element()))
    ocurrencias ++;
  for (Position<E> v: tree.children(node)) {
    ocurrencias += ocurrenciasRec(tree, v, elem);
  }
  return ocurrencias;
}
```

Más compacto:

```
public static <E> int ocurrenciasRec(BinaryTree<E> tree,
                                     Position<E> node, E elem) {
  return
    (eqNull(elem,node.element()) ? 1: 0) +
    (tree.hasLeft(node)  ? ocurrenciasRec(tree, tree.left(node), elem) : 0) +
    (tree.hasRight(node) ? ocurrenciasRec(tree, tree.right(node), elem): 0);
}
```

Ejercicio 131 (solución propuesta)

```
public static <E> String toStringExp(BinaryTree<E> tree) {
  return toStringExpRec(tree, tree.root());
}

public static <E> String toStringExpRec(BinaryTree<E> tree,
                                        Position<E> node) {

  if (tree.isExternal(node)) return node.element().toString();

  String s = "(";
  s += toStringExpRec(tree,tree.left(node));
  s += node.element().toString();
  s += toStringExpRec(tree,tree.right(node));
  s += ")";
  return s;
}
```

Ejercicio 132 (solución propuesta)

Apartado (a):

- L9, es necesario invertir el orden del **`if`** para evitar posibles `NullPointerException`

  ```
  if (tree.parent(node)!=null &&  node.element()<tree.parent(node).element())
  ```

- L17, es necesario recoger el valor de la llamada recursiva en la variable `res`

  ```
  res = heap (tree,tree.right(node));
  ```

Apartado (b):

- L18, es necesario considerar el valor de `res` para no recorrer la rama derecha

  ```
  if (res && tree.hasRight(node))
  ```

Este sería un posible código que cumple lo pedido. El código en negrita se añade o modifica el código anterior para tener código correcto y eficiente.

```
public static boolean hasHeapProperty (BinaryTree<Integer> tree) {
  return tree.isEmpty() || hasHeapProperty(tree,tree.root());
}

public static boolean hasHeapProperty (BinaryTree<Integer> tree,
                                       Position<Integer> node) {

  if (tree.parent(node) != null &&
      node.element() < tree.parent(node).element()}) {
     return false;
  }

  boolean res = true;
  if (tree.hasLeft(node)) {
     res = hasHeapProperty (tree,tree.left(node));
  }
  if (res && tree.hasRight(node)) {
     res = hasHeapProperty (tree,tree.right(node));
  }
  return res;
}
```

Ejercicio 133 (solución propuesta)

```
  void printWordsInTrie (Tree<Pair<Character,Boolean>> tree) {
    printWordsInTrie(tree,tree.root(), "");
  }

  void printWordsInTrie (Tree<Pair<Character,Boolean>> tree,
                         Position<Pair<Character,Boolean>> node,
                         String path) {
    path += node.element().getLeft();
    if (node.element().getRight()) {
      System.out.println(path);
    }
    for (Position<Pair<Character,Boolean>> child: tree.children(node)) {
      printWordsInTrie(tree, child, path);
    }
  }
```

Ejercicio 134 (solución propuesta)

```
PositionList<Integer> hojasEnRango(BinaryTree<Integer> tree,
                                   int a, int b ) {
  if (tree == null) {
    throw new IllegalArgumentException();
  }
  PositionList<Integer> res = new NodePositionList<>();

  hojasEnRango(tree,a,b,tree.root(),res);
}

private static void hojasEnRango(BinaryTree<Integer> tree,
                                 int a, int b,
                                 Position<Integer> v,
                                 PositionList<Integer> res) {
  if (tree.isExternal(v) && v.element() != null &&
      v.element() >= a && v.element() <= b) {
    res.addLast(v.element());
  }
  if (tree.hasLeft(v)) {
    hojasEnRango(tree,a,b,tree.left(v),res);
  }
  if (tree.hasRight(v)) {
    hojasEnRango(tree,a,b,tree.left(v),res);
  }
}
```

Ejercicio 135 (solución propuesta)

```
public static int sumaNodos2Hijos(BinaryTree<Integer> tree) {
  if (tree == null) {
    throw new IllegalArgumentException();
  }
  if (tree.isEmpty()) {
    return 0;
  }

  return sumaNodos2Hijos(tree,tree.root());
}

private static int sumaNodos2Hijos(BinaryTree<Integer> tree,
                                   Position<Integer> v) {
  int suma = 0;
  if (tree.hasLeft(v) && tree.hasRight(v) && v.element() != null ) {
    suma += v.element();
  }
  if (tree.hasLeft(v)) {
    suma += sumaNodos2Hijos(tree,tree.left(v));
  }
  if (tree.hasRight(v)) {
    suma += sumaNodos2Hijos(tree,tree.right(v));
  }

  return suma;
}
```

Ejercicio 136 (solución propuesta)

(a)
- La implementación con array de colas FIFO aquí sugerida: respeta la antigüedad dado que la cola correspondiente a los elementos con una prioridad dada es una estructura FIFO.
- Una implementación basda en un montículo: no, no respeta la antigüedad. Por ejemplo supón que en un montículo vacío se insertan tres elementos `e1`, `e2` y `e3` (en este orden), todos con la misma prioridad. Si a continuación ejecutamos tres veces la operación `dequeue()` extraeremos los elementos en orden `e1`, `e3` y `e2`.

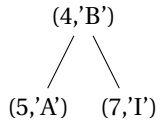
(b) **`enqueue(K clave, V valor)`** $O(n)$, siendo n el número total de pares en la cola con prioridad. Peor caso: todos los pares están en la cola correspondiente a esa prioridad (clave).

`dequeue()` $O(1)$

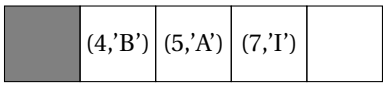
`first()` $O(1)$

Ejercicio 137 (solución propuesta)

(a)

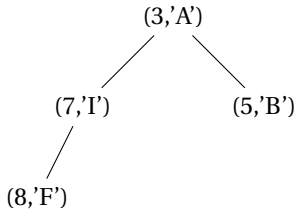


(b)

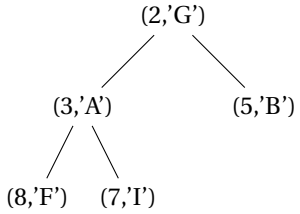


Ejercicio 138 (solución propuesta)

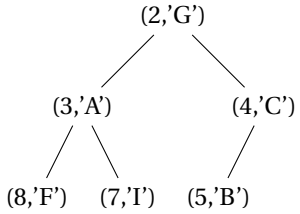
Las cuatro primeras entradas se insertan en el árbol (casi)completo:



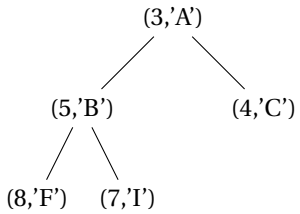
La inserción de (2,'G') produce «up-heap bubbling»:



La inserción de (4,'C') produce «up-heap bubbling»:

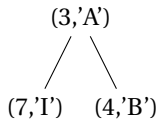


El borrado de la clave mínima produce «down-heap bubbling»:

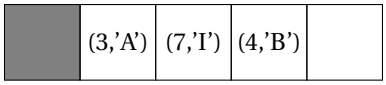


Ejercicio 139 (solución propuesta)

(a)



(b)



Ejercicio 140 (solución propuesta)

(a) La lista está ordenada. Se devuelve el primer elemento asumiendo que es el mínimo. Si fuese una lista arbitraria la entrada de menor clave podría estar en cualquier posición de la lista.

(b) En una cola con prioridad donde se guardan pares clave-valor, las claves determinan la prioridad, con la menor clave indicando la mayor prioridad. Si se desea invertir la prioridad de las entradas, lo correcto es reasignarles nuevas claves o redefinir el comportamiento del comparador (método `compare`) para el tipo clave. En la propuesta mencionada no se adaptan el resto de métodos. La posición de la entrada con más prioridad en una lista ordenada puede estar al final siempre que la inserción y la búsqueda se comporten consistentemente con esta decisión de diseño.

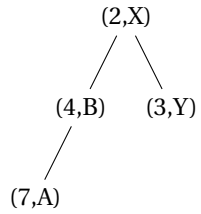
Ejercicio 141 (solución propuesta)

Un *heap* es un árbol binario en el que cada nodo almacena entradas y cumple las siguientes propiedades:

- *Heap-Order Property*: Para cada nodo interno v que no es la raíz se cumple que la clave almacenada en v es mayor o igual que la clave almacenada en el padre de v.
- *(Almost)Complete Binary Tree Property:* Sea h la altura del árbol.
  - Hay 2^i nodos en el nivel i, para i de 0 a $h-1$.
  - En el nivel $h-1$ todos los nodos internos están a la izquierda de los externos, habiendo como máximo un nodo con un hijo que debe ser el izquierdo

Ejercicio 142 (solución propuesta)

(a)

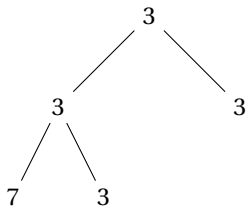


(b)

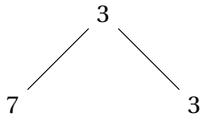
(2,X), (3,Y), (4,B), (7,A)

Ejercicio 143 (solución propuesta)

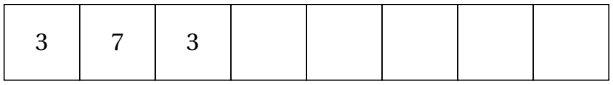
(a)



(b)

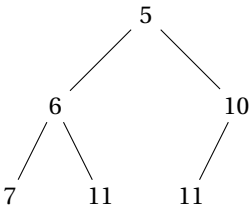


(c)

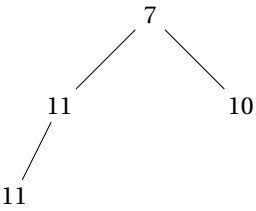


Ejercicio 144 (solución propuesta)

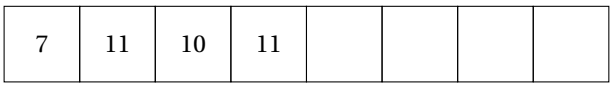
(a)



(b)

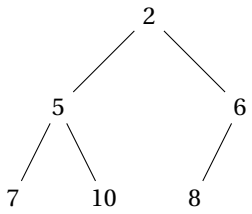


(c)

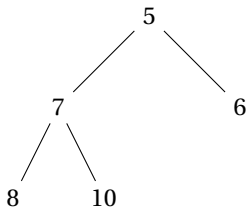


Ejercicio 145 (solución propuesta)

(a)

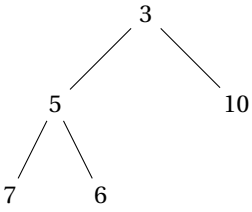


(b)

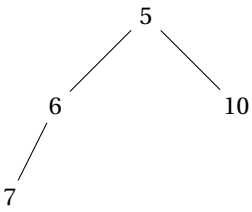


Ejercicio 146 (solución propuesta)

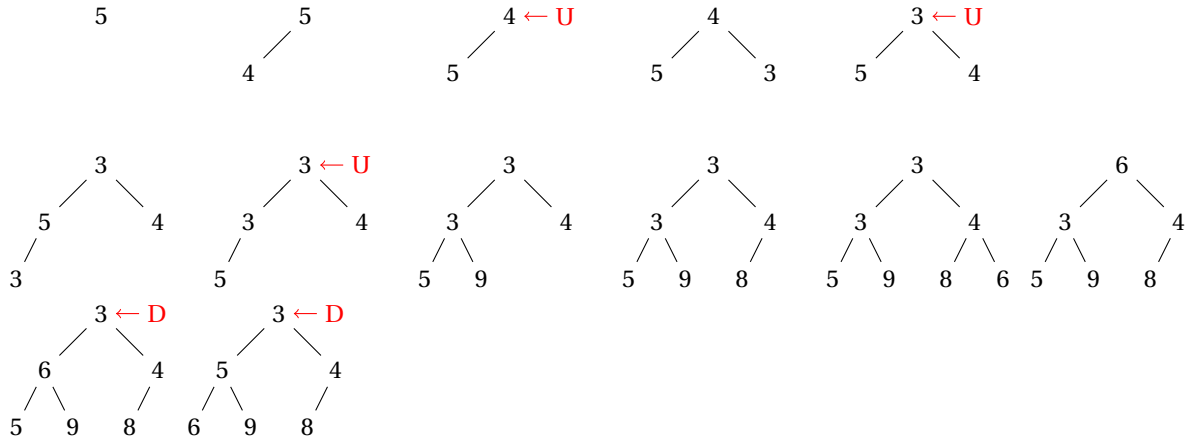
(a)



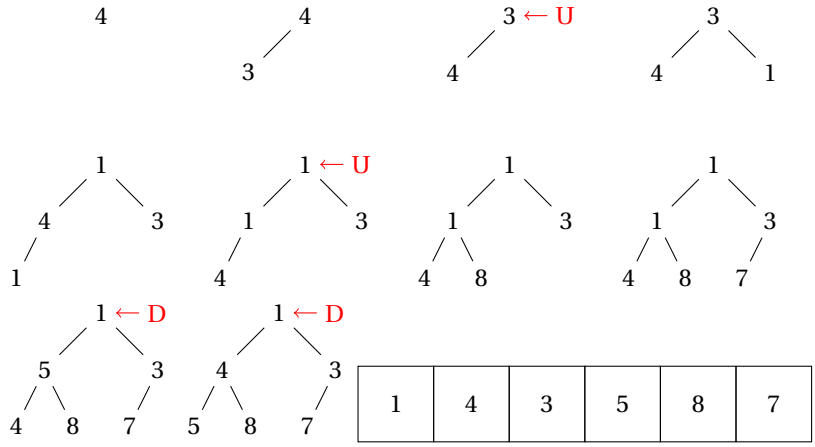
(b)



Ejercicio 147 (solución propuesta)



Ejercicio 148 (solución propuesta)



Ejercicio 149 (solución propuesta)

```
PositionList<Entry<K,V>> list = new NodePositionList<Entry<K,V>>();
if (pq != null && !pq.isEmpty()) {
  list.addFirst(pq.dequeue());
  while (!pq.isEmpty() &&
         pq.first().getKey().equals(list.first().element().getKey()))
    list.addLast(pq.dequeue());
}
return list;
```

Ejercicio 150 (solución propuesta)

(a)
```
public static Alumno [] ordenarAlumnos (Alumno [] alumnos) {
    Alumno [] resultado = new Alumno [alumnos.length];
    PriorityQueue<Integer,Alumno> pq;
    pq = new SortedListPriorityQueue<Integer,Alumno>();

    for (int i = 0; i < alumnos.length; i ++) {
      pq.enqueue(alumnos[i].getDni(),alumnos[i]);
    }

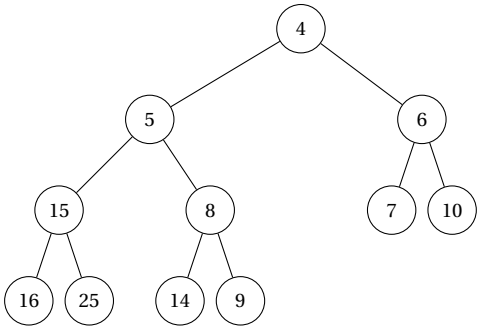
    int i = 0;
    while(!pq.isEmpty()) {
      resultado[i] = pq.dequeue().getValue();
      i ++;
    }

    return resultado;
  }
```

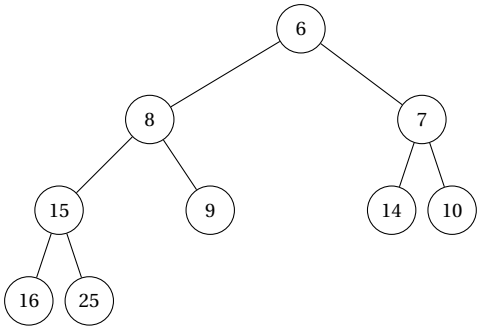
(b)
```
public int compareTo(Alumno o) {
    return this.dni - o.getDni();
  }
```

Ejercicio 151 (solución propuesta)

(a)



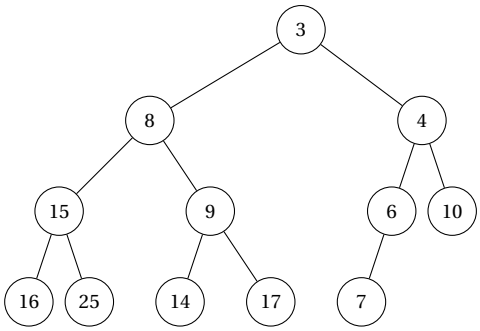
(b)



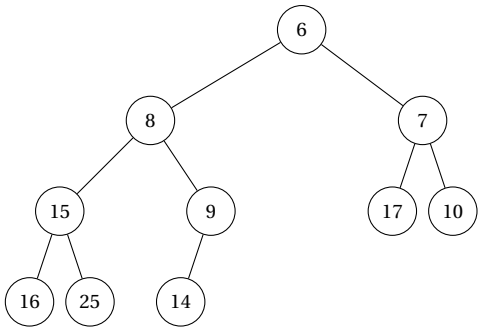
Ejercicio 152 (solución propuesta)

Solución:

Apartado (a)

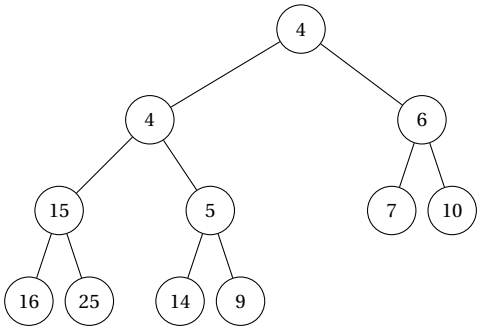


Apartado (b)

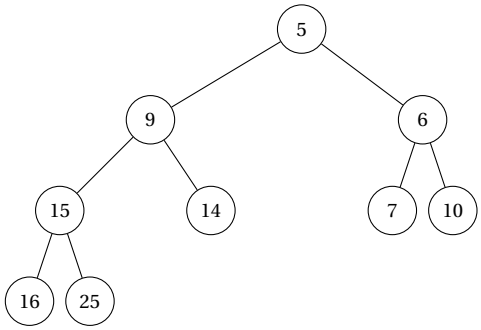


Ejercicio 153 (solución propuesta)

Apartado (a)



Apartado (b)



Ejercicio 154 (solución propuesta)

1. Usar el operador de igualdad ==.
2. Usar el método `equals()`.

Ejercicio 155 (solución propuesta)

```
public V get(K k) {
  Entry<K,V> e;
  for ( Iterator<Entry<K,V>> it = lista.iterator() ;
        it.hasNext() && !(e = it.next().element()).getKey().equals(k) ;
      ) ;
  return e.getKey().equals(k) ? e.getValue() : null ;
}
```

Ejercicio 156 (solución propuesta)

```
public static Map<Character,Integer> contarApariciones(String texto) {
  Map<Character,Integer> map = new HashTableMap<Character,Integer>();

  for(int i = 0; i < texto.length(); i ++) {
    Integer n = map.get(texto.charAt(i));
    map.put(texto.charAt(i),(n == null? 1 : n + 1));
  }

  return map;
}
```

Ejercicio 157 (solución propuesta)

Solución:

```
public static void aumentarPrecio (Map<String,Double> precios,
                                   String prod, Double coste) {

  Double precio = precios.get(prod);
  precio = (precio != null ? coste + precio : coste);
  precios.put(prod,precio);
}
```

Ejercicio 158 (solución propuesta)

Solución:

```
public static Map<Integer,Integer> contarPracticas(PositionList<Alumno> lista) {
  Map<Integer,Integer> map = new HashTableMap<>();

  for(Alumno alumno: lista) {
    Integer nprac = map.get(alumno.getDni());
    map.put(alumno.getDni(), (nprac == null ? 1: nprac +1));
  }

  return map;
}
```

Ejercicio 159 (solución propuesta)

```
public static Map<String,String> invertir (Map<String,String> map) {
  Map<String,String> mapres = new HashTableMap<>();
  Iterator<Entry<String,String>> it = map.entries();
  while(it.hasNext()) {
    Entry<String,String> entry = it.next();
    mapres.put(entry.getValue(), entry.getKey());
  }
  return mapres;
}
```

Ejercicio ?? (solución propuesta)

El método cuenta el número de apariciones de cada uno de los caracteres del `String` que se recibe como parámetro. El orden de salida puede variar en diferentes ejecuciones al estar usándose un `HashTableMap`. Una posible salida sería:

```
(i - 2),(m - 2),(o - 1),(? - 1),(p - 1),(r - 1),
```

Ejercicio 161 (solución propuesta)

```
public class MiSet<E> implements Set<E> {

  private Map<E,E> elements;

  @Override
  public boolean isEmpty() {
    return elements.isEmpty();
  }

  @Override
  public int size() {
    return elements.size();
  }

  @Override
  public boolean add(E elem) {
    if (elem == null) {
      throw new IllegalArgumentException();
    }
    E old = elements.put(elem, elem);
    return old != null;
  }

  @Override
  public boolean remove(E elem) {
    if (elem == null) {
      throw new IllegalArgumentException();
    }
    E old = elements.remove(elem);
    return old != null;
  }

  @Override
  public boolean contains(Object o) {
    return elements.containsKey(o);
  }

  @Override
  public Iterable<E> getElements() {
    return elements.keys();
  }

  @Override
  public Set<E> intersection (Set<E> set) {
    Set<E> res = new MiSet<>();

    for (E e: this.getElements()) {
      if (set.contains(e)) {
        res.add(e);
      }
    }
    return res;
  }
}
```

Ejercicio 162 (solución propuesta)

```
PositionList<String> getAlumnosEnRango (Map<String,Integer> map, int a, int b) {
  PositionList<String> res = new NodePositionList<>();
  Iterator<Entry<String,Integer>> it = map.entries().iterator();
  while (it.hasNext()) {
    Entry<String,Integer> e = it.next();
    if (e.getValue() >= a && e.getValue() <= b) {
      res.addLast(e.getKey());
    }
  }
  return res;
}
```

Ejercicio 163 (solución propuesta)

```
public static PositionList<String> ordenarPorNota (Map<String,
                                                    Integer> map) {

  PriorityQueue<Integer, String> pq = new HeapPriorityQueue<>();
  PositionList<String> res = new NodePositionList<>();

  for (Entry<String,Integer> e: map.entries()) {
    pq.enqueue(e.getValue(), e.getKey());
  }

  while (!pq.isEmpty()) {
    res.addLast(pq.dequeue().getValue());
  }
  return res;
}
```

Ejercicio 164 (solución propuesta)

```
public static <E> PositionList<E> ordenarDescPorApariciones
                                        (Map<E, Integer> map) {

  PriorityQueue<Integer, E> pq = new HeapPriorityQueue<>();
  PositionList<E> res = new NodePositionList<>();

  for (Entry<E,Integer> e: map.entries()) {
    pq.enqueue(e.getValue(), e.getKey());
  }

  while (!pq.isEmpty()) {
    res.addFirst(pq.dequeue().getValue());
  }
  return res;
}
```

Ejercicio 165 (solución propuesta)

$O(1) \subset O(\log_5 n) = O(\log_e n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^4) \subset O(2^n) \subset O(n!)$

Ejercicio 170 (solución propuesta)

$O(n)$

Ejercicio 171 (solución propuesta)

$O(n^3)$

Ejercicio 172 (solución propuesta)

A C B E D

Ejercicio 173 (solución propuesta)

(c)

Ejercicio 174 (solución propuesta)

C E B A D

Ejercicio 175 (solución propuesta)

La complejidad de recorrer completamente un vector es $O(n)$ con n el tamaño del vector. Sin embargo, $n \leq 20$, es decir, el tamaño del vector está acotado. En el caso peor la complejidad será $O(20)$ que es lo mismo que $O(1)$, es decir, complejidad constante.

Ejercicio 176 (solución propuesta)

La inserción en una lista ordenada tiene complejidad lineal. Para cada índice i de 0 hasta n (tamaño del vector) se inserta el elemento $v[i]$ en la lista de tamaño i de forma ordenada. La complejidad es la suma de las complejidades de las inserciones: $1+2+\ldots+n=(n^2-n)/2$ que es $O(n^2)$.

Ejercicio 177 (solución propuesta)

Esta pregunta es ambigua. La complejidad de `listaMatriculados.member(it.next())` es $O(1)$ pues el tamaño m de `listaMatriculados` es una constante. Si $n \leq m$ entonces n está acotado por una constante y por tanto la complejidad de `matriculados` es $O(1)$. Pero debemos considerar el caso peor $n > m$. El bucle **`while`** termina cuando se ha recorrido `lista` o al encontrar un alumno de `lista` que no está en `listaMatriculados`. Si todos los alumnos en `lista` están en `listaMatriculados` entonces para que $n > m$ se cumpla tiene que haber alumnos repetidos en `lista`. Si hay alumnos repetidos entonces la complejidad es $O(n)$. Si no los hay, entonces $n \leq m$ y la complejidad es $O(1)$ porque no puede cumplirse $n > m$ y que todos los alumnos de `lista` estén en `listaMatriculados` sin haber alumnos repetidos. Es irrelevante si `listaMatriculados` tiene alumnos repetidos. El caso peor es aquel en el que hay alumnos repetidos y la complejidad es $O(n)$. Sin embargo el enunciado no dice nada sobre alumnos repetidos y podría suponerse que $n \leq m$ es el caso en el contexto de la pregunta.

Ejercicio 178 (solución propuesta)

La complejidad en caso peor será $O(n)$ donde n será el tamaño de la lista. El iterador `NonNullElementIterator` itera sobre todos los nodos de la lista con el iterador ordinario.

Ejercicio 179 (solución propuesta)

En el caso peor la lista de tamaño n tiene n apariciones del elemento que debe saltarse el iterador. La complejidad del método `next()` en ese caso es $O(n)$ pues recorre toda la lista hasta dejar el cursor a null.

Ejercicio 180 (solución propuesta)

- El método `m1` tiene complejidad $O(1)$.
- El método `m2` tiene complejidad $O(n^2)$, siendo n la longitud de la lista `l`.

Ejercicio 181 (solución propuesta)

- El método `m1` tiene complejidad $O(n^2)$
- El método `m2` tiene complejidad $O(n)$

Ejercicio 182 (solución propuesta)

- El método `m1` tiene complejidad $O(n)$
- El método `m2` tiene complejidad $O(n^2)$

Ejercicio 183 (solución propuesta)

- El método `m1` tiene complejidad $O(1)$
- El método `m2` tiene complejidad $O(n^2)$

Ejercicio 184 (solución propuesta)

- El método `m1` tiene complejidad $O(n)$
- El método `m2` tiene complejidad $O(n)$ (también valdría $O(2 * n)$)

Ejercicio ?? (solución propuesta)

Solución:

- El método `method1` tiene complejidad $O(n)$
- El método `method2` tiene complejidad $O(log(n))$

Ejercicio 186 (solución propuesta)

- El método `method1` tiene complejidad $O(n^2)$, donde n es la longitud de la lista `list`.
- El método `method2` tiene complejidad $O(n * log(n))$, donde n es la longitud de la lista `l`.

Ejercicio 187 (solución propuesta)

- El método `method1` tiene complejidad $O(n)$
- El método `method2` tiene complejidad $O(log(n))$

Ejercicio 188 (solución propuesta)

- El método `method1` tiene complejidad $O(n)$
- El método `method2` tiene complejidad $O(n^2)$

Ejercicio 189 (solución propuesta)

- L14, el método debe devolver **`true`** cuando ambos nodos sean iguales.

```
return @\textbf{true}@;
```

- L20, es necesario parar el bucle cuando el valor de `reachable` tome valor **`true`** después de la llamada recursiva.

```
while (it.hasNext() && @\textbf{!reachable}@) {
```

- L23, es necesario recoger el valor de la llamada recursiva a `isReachable` para poder saber si ha habido éxito en la búsqueda.

```
@\textbf{reachable = }@isReachable(g, other, to, visited);
```

Este sería un posible código que cumple lo pedido. El código en negrita se añade o modifica el código anterior para tener código correcto y eficiente.

```
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to) {
  Set<Vertex<V>> visited = new HashTableMapSet<Vertex<V>>();
  return isReachable(g, from, to, visited);
}

public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to,
                                         Set<Vertex<V>> visited ) {

  if (from == to) {
    return @\textbf{true}@;
  }

  visited.add(from);
  boolean reachable = false;
  Iterator<Edge<E>> it = g.edges(from).iterator();
  while (it.hasNext() && @\textbf{!reachable}@) {
    Vertex<V> other = g.opposite(from, it.next());
    if (!visited.contains(other)) {
      @\textbf{reachable = }@isReachable(g, other, to, visited);
    }
  }
  return reachable;
}
```

Ejercicio 190 (solución propuesta)

```
static <V,E> Set<Vertex<V>> getVerticesAlcanzables (UndirectedGraph<V, E> g,
                                                    Vertex<V> n) {
  Set<Vertex<V>> visited = new HashTableMapSet<V>();
  getVerticesAlcanzablesRec(g,n,visited);
  return visited;
}
static <V,E> void getVerticesAlcanzablesRec (UndirectedGraph<V, E> g,
                                             Vertex<V> n,
                                             Set<Vertex<V>> visited ) {
  if (visited.contains(n)) {
    return;
  }
  visited.add(n);
  for (Edge<E> e: g.edges(n)) {
    getVerticesAlcanzablesRec(g, g.opposite(n, e), visited);
  }
}
```

Ejercicio 191 (solución propuesta)

```
  public static <V,E> boolean isReachableInNSteps (DirectedGraph<V, E> g,
                                                   Vertex<V> from,
                                                   Vertex<V> to,
                                                   int n) {
    return isReachableInNSteps(g, from, to, n);
  }

  public static <V,E> boolean isReachableInNStepsRec (DirectedGraph<V, E> g,
                                                      Vertex<V> from,
                                                      Vertex<V> to,
                                                      int n) {

   if (from == to) {
     return true;
   }
   if  (n >= 0) { // ERROR -> if (n <= 0)
     return false;
   }

   boolean reachable = false;
   Iterator<Edge<E>> it = g.edges(from).iterator();
   // ERROR -> g.outgoingEdges(from).iterator()
   while (it.hasNext()) { // ERROR -> && while (it.hasNext() && !reachable)
     reachable = isReachableInNStepsRec(g, g.endVertex(from), to, n, visited);
     // ERROR -> reachable = isReachableInNSteps(g, g.endVertex(it.next()), to, n-1);
   }
   return reachable;
 }
```

Ejercicio 192 (solución propuesta)

```
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to) {
 Set<Position<?>> visited = new HashTableMapSet<>();
 return isReachable(g, from, to, visited); // ERROR enunciado
}
public static <V,E> boolean isReachable (UndirectedGraph<V, E> g,
                                         Vertex<V> from,
                                         Vertex<V> to,
                                         Set<Position<?>> visited ) {

  if (from == to) {
    return true;
  }
  if  (visited.contains(from)) {
    return false; // ERROR
  }

  visited.add(from); // ERROR
  boolean reachable = false;
  Iterator<Edge<E>> it = g.edges(from).iterator();
  while (it.hasNext() && !reachable) { // ERROR
    reachable = isReachable(g, g.opposite(from, it.next()), to, visited); // ERROR
  }
  return reachable;
}
```

```
<V,E> PositionList<Vertex<V>> reachableNodesWithGrade (DirectedGraph<V, E> g,
                                                       Vertex<V> v,
                                                       int grade) {
  Set<Vertex<V>> visited  = new HashTableMapSet<>();
  PositionList<Vertex<V>> res = new NodePositionList<>();

  reachableNodesWithGrade(g, v, grade, visited, res);
  return res;
}

private static <V,E> void reachableNodesWithGrade (DirectedGraph<V, E> g,
                                                   Vertex<V> v,
                                                   int grade,
                                                   Set<Vertex<V>> visited,
                                                   PositionList<Vertex<V>> res) {

  if (visited.contains(v)) {
    return;
  }
  if (g.outDegree(v) == grade) {
    res.addLast(v);
  }
  visited.add(v);
  for (Edge<E> e: g.outgoingEdges(v)) {
    reachableNodesWithGrade(g, g.endVertex(e), grade, visited, res);
  }
}
```

Ejercicio 194 (solución propuesta)

```
public static <V,E> int numReachableInSteps (UndirectedGraph<V, E> g,
                                             Vertex<V> v,
                                             int steps) {

  Set<Vertex<?>> visited = new HashTableMapSet<>();
  visited.add(v);
  numReachableInSteps(g,v,visited,steps);
  return visited.size();
}

public static <V,E> void numReachableInSteps (UndirectedGraph<V, E> g,
                                              Vertex<V> v,
                                              Set<Vertex<?>> visited,
                                              int steps) {

  if (steps == 0 || visited.contains(v)) {
    return;
  }
  visited.add(v);
  for (Edge<E> e: g.edges(v)) {
    numReachableInSteps(g, g.opposite(v, e), visited, steps - 1);
  }
}
```

A. Interfaces y clases

A.1. Interfaz `Iterable<E>`

```
public interface Iterable <E> {
  /** Returns an iterator over a set of elements of type T. **/
  public Iterator<E> iterator();
}
```

A.2. Interfaz `Iterator<E>`

```
public interface Iterator<E> {
  /** Returns the next element in the iteration. */
  public E next();
  /** Returns true if the iteration has more elements.  */
  public boolean hasNext();
}
```

A.3. Interfaz `PositionList<E>`

```
public interface PositionList<E> extends Iterable<E> {
  /** Returns the number of nodes in the list. */
  public int size();
  /** Returns true if the list is empty (has no nodes), false if it's not. */
  public boolean isEmpty();
  /** Returns the first node of the list, or null if the list is empty. */
  public Position<E> first();
  /** Returns the last node of the list, or null if the list is empty. */
  public Position<E> last();
  /** Either returns the next node to the right of parameter node 'p' in the
   *  list, or returns null if 'p' doesn't have a next node to the right.
   *  Raises the exception if 'p' is null or is not a node of the list. */
  public Position<E> next(Position<E> p) throws IllegalArgumentException;
  /** Either returns the previous node to the left of parameter node 'p' in
   *  the list, or returns null if 'p' doesn't have a previous node to the
   *  left. Raises the exception if 'p' is null or is not a node of the list. */
  public Position<E> prev(Position<E> p) throws IllegalArgumentException;
  /** Inserts a new first node to the list with 'elem' as element. */
  public void addFirst(E elem);
  /** Inserts a new last node to the list with 'elem' as element.  */
  public void addLast(E elem);
  /** Inserts a new node with 'elem' as element to the list right before
   *  parameter node 'p'. Raises the exception if 'p' is null or is not a node
   *  of the list.  */
  public void addBefore(Position<E> p, E elem) throws IllegalArgumentException;
  /** Inserts a new node with 'elem' as element to the list right after
   *  parameter node 'p'. Raises the exception if 'p' is null or is not a node
   *  of the list. */
  public void addAfter(Position<E> p, E elem) throws IllegalArgumentException;
  /** Removes node 'p' from the list and returns its element. Raises the
   *  exception if 'p' is null or is not a node of the list.  */
  public E remove(Position<E> p) throws IllegalArgumentException;
  /** Sets the element of node 'p' on the list to 'elem' and returns the old
   *  element in 'p'. Raises the exception if 'p' is null or is not a node of
```

```
   *  the list.  */
  public E set(Position<E> p, E elem) throws IllegalArgumentException;
  /** Returns an array with all the elements of the list or an empty array
   *  of length zero if the list is empty.  */
  public Object [] toArray();
}
```

A.4. Interfaz `Position<E>`

```
public interface Position<E> {
  /* Returns the element contained in the node */
  public E element();
}
```

A.5. Clase `NodeList<E>`

```
public class NodeList<E> implements Position<E> {
  private Node<E> prev;
  private Node<E> next;
  private E element;

  public Node(Node<E> newPrev, Node<E> newNext, E elem) {
    prev = newPrev;
    next = newNext;
    element = elem;
  }

  public E element() { return element; }

  public Node<E> getNext() { return next; }
  public Node<E> getPrev() { return prev; }

  public void setNext(Node<E> newNext) { next = newNext; }
  public void setPrev(Node<E> newPrev) { prev = newPrev; }
  public void setElement(E newElement) { element = newElement; }
}
```

A.6. Interfaz `FIFO<E>`

```
public interface FIFO<E> extends Iterable<E> {
  /** Returns the number of elements. */
  public int size();
  /** Returns true if empty */
  public boolean isEmpty();
  /** Returns first element in the FIFO if non-empty, otherwise throws
   *  the exception. */
  public E first() throws EmptyFIFOException;
  /** Enqueues the element as the last element of FIFO.  */
  public void enqueue(E elem);
  /** Dequeues and returns the first element in the FIFO if non-empty,
    * otherwise throws the exception.  */
  public E dequeue() throws EmptyFIFOException;
  /** Returns an array with all the elements of the FIFO */
  public Object [] toArray();
```

```
}
```

A.7. Interfaz `LIFO<E>`

```
public interface LIFO<E> extends Iterable<E> {
  /** Returns the number of elements. */
  public int size();
 /** Returns true if empty */
  public boolean isEmpty();
  /** Returns the element on top of the LIFO if non-empty, otherwise throws
   *  the exception.*/
  public E top() throws EmptyStackException;
  /** Removes and returns the element on top of the LIFO if non-empty,
    * otherwise throws the exception.   */
  public E pop() throws EmptyStackException;
  /** Pushes the element on top of the LIFO.   */
  public void push(E elem);
  /** Returns an array with all the elements of the LIFO */
  public Object [] toArray();
}
```

A.8. Interfaz `Comparable<T>`

```
public interface Comparable<T> {
  /** Returns a negative integer, zero, or a positive integer when this
    * object is less than, equal to, or greater than the object o.*/
  public int compareTo (T o);
}
```

A.9. Interfaz `Comparator<T>`

```
public interface Comparator<T> {
  /** Returns a negative integer, zero, or a positive integer when o
    * object is less than, equal to, or greater than the object o2.*/
  public int compare (T o, T o2);
}
```

A.10. Interfaz `Tree<E>`

```
public interface Tree<E> extends Iterable<E> {
  /** Returns the size of the tree. */
  public int size();
  /** Returns true if the tree is empty, and false otherwise. */
  public boolean isEmpty();
  /** Stores a new value e in the tree position specified by p  and returns
   * the previous value. */
  public E set(Position<E> p, E e) throws IllegalArgumentException;
  /** Returns the root position of the tree. */
  public Position<E> root();
  /**  Returns the parent position of p, or null if p has no parent (is the root) */
  public Position<E> parent(Position<E> p) throws IllegalArgumentException;
  /** Returns true if the position parameter p is an internal node */
  public boolean isInternal(Position<E> p);
  /** Returns true if the position parameter p is an external node /
```

```
  public boolean isExternal(Position<E> p);
  /** Returns true if the position p is the root node of the tree */
  public boolean isRoot(Position<E> p);
  /** Adds a root node to the tree, whose position has the value e */
  public Position<E> addRoot(E e) throws NonEmptyTreeException;
  /** Returns an iterator over the elements in positions in the tree.*/
  public Iterator<E> iterator();
  /** Returns an iterable class over the positions of the children */
  public Iterable<Position<E>> children(Position<E> p);
```

A.11. Interfaz `GeneralTree<K,V>`

```
public interface GeneralTree<E> extends Tree<E> {
  /** Adds a new node to a tree containing the value e. The new node is inserted
    * as the first (leftmost) child of the node at position parentPos. Returns the
    * new position */
  public Position<E> addChildFirst(Position<E> parentPos, E e);
  /** Adds a new node to a tree containing the value e. The new node is inserted
    * as the last (rightmost) child of the node at position parentPos. Returns the
    * new position*/
  public Position<E> addChildLast(Position<E> parentPos, E e);
  /** Adds a new node to a tree containing the value e. The new node is inserted
   * immediately to the right of the node at position siblingPos. Returns the new
   *  position. */
  public Position<E> insertSiblingBefore(Position<E> siblingPos, E e);
  /** Adds a new node to a tree containing the value e. The new node is inserted
   *  immediately to the left of the node at position siblingPos. Returns the new
   * position. */
  public Position<E> insertSiblingAfter(Position<E> siblingPos, E e);
  /** Removes the subtree rooted in the position argument p from the tree.*/
  public void removeSubTree(Position<E> p);
```

A.12. Interfaz `Entry<K,V>`

```
public interface Entry<K,V> {
    /** Returns the entry key. */
  public K getKey();
    /** Returns the entry value. */
  public V getValue();
}
```

A.13. Interfaz `BinaryTree<E>`

```
public interface BinaryTree<E> extends Tree<E> {
  /** Returns true if the node at position p has a child to the left. */
  public boolean hasLeft(Position<E> p);
  /** Returns true if the node at position p has a child to the right. */
  public boolean hasRight(Position<E> p);
  /** Returns the element of the left child of the node at position
   * p, or null if there is no such child. */
  public Position<E> left(Position<E> p);
  /** Returns the element of the right child of the node at position
   *  p, or null if there is no such child. */
  public Position<E> right(Position<E> p);
```

```
  /** Inserts a new child with parent at position p, to the left. Throws
    * NodeAlreadyExistsException if there is already a left child of position
    *  p. Returns the new position */
  public Position<E> insertLeft(Position<E> parentPos, E e)
         throws NodeAlreadyExistsException;
  /** Inserts a new child with parent at position p, to the right. Throws
    * NodeAlreadyExistsException if there is already a right child of position
    *  p. Returns the new position */
  public Position<E> insertRight(Position<E> parentPos, E e)
        throws NodeAlreadyExistsException;
  /** Removes from the tree the subtree rooted in the position argument p */
  public void removeSubTree(Position<E> p);
}
```

A.14. Interfaz `PriorityQueue<K,V>`

```
public interface PriorityQueue<K,V> {
  /** Returns the number of items in the priority queue. */
  public int size();
  /** Returns whether the priority queue is empty. */
  public boolean isEmpty();
  /** Returns but does not remove an entry with minimum key. */
  public Entry<K,V> first() throws EmptyPriorityQueueException;
  /** Inserts a key-value pair and return the entry created. */
  public Entry<K,V> enqueue(K key, V value) throws InvalidKeyException;
  /** Removes and returns an entry with minimum key. */
  public Entry<K,V> dequeue() throws EmptyPriorityQueueException;
}
```

A.15. Interfaz `Map<K,V>`

```
public interface Map<K,V> {
  /** Returns the number of items in the map. */
  public int size();
  /** Returns whether the map is empty. */
  public boolean isEmpty();
  /** Puts a given key-value pair in the map, replacing a previous
   * one, if any, and returns the old value. */
  public V put(K key, V value);
  /** Returns the value associated with a key or null if key does not exist */
  public V get(K key);
  /** Removes the key-value pair with a given key. */
  public V remove(K key);
  /** Returns an iterable object containing all the keys in the map. */
  public Iterable<K> keySet();
  /** Returns an iterable object containing all the values in the map. */
  public Iterable<V> values();
  /** Returns an iterable object containing all the entries in the map. */
  public Iterable<Entry<K,V>> entrySet();
  /** Returns true if the map contains an entry with the key argument,
      and false otherwise. */
  public boolean containsKey(Object key) throws InvalidKeyException;

}
```