

**EI1013/MT1013 ESTRUCTURA DE DATOS**  
**BOLETÍN DE PROBLEMAS III**  
**IMPLEMENTACIÓN DE ESTRUCTURAS DE DATOS**  
**LINEALES**

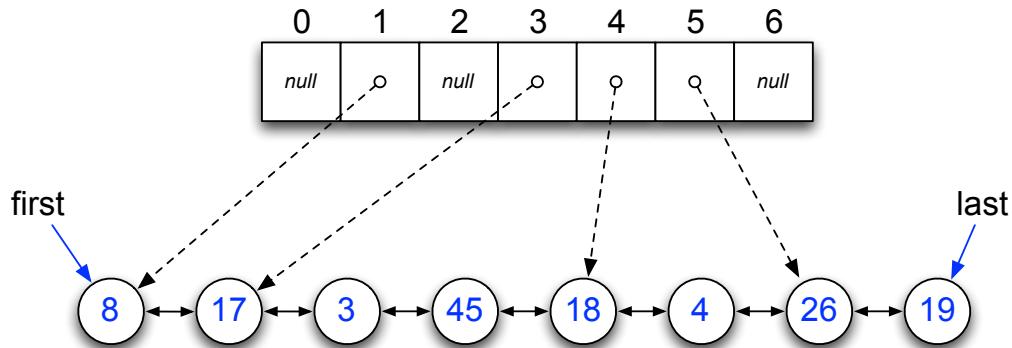
*En este boletín se recopilan ejercicios de examen. Todos los ejercicios han aparecido en algún examen final de la asignatura. Éste que se indica en la línea final del enunciado. Además, en cada ejercicio se indica el peso que tuvo ese ejercicio sobre la nota máxima del examen, siempre 10 puntos. Para una parte de los ejercicios se incluyen posibles soluciones.*

- 3.1 (3 puntos)** Deseamos implementar un variante del tipo de datos conjunto que admite elementos repetidos. Para ello decidimos usar una tabla de dispersión para almacenar los elementos. Las colisiones se resolverán mediante el método de direccionamiento abierto. Tendrá la peculiaridad de que por cada elemento almacenado en la tabla se guardará una cuenta con el número de copias que aparece en el conjunto. De tal manera que la primera vez que se guarda un elemento en el conjunto el número de la cuenta será 1. Si vuelve a guardarse un segunda copia del mismo elemento la cuenta se incrementará a 2.

```
public class HashBag<E> implements Set<E> {  
    private ArrayList<E> table;  
    private ArrayList<Integer> count;  
    private int size;  
    private int used;  
  
    private void rehash();  
    ...  
}
```

- **table** almacena los datos de la tabla de dispersión. Tiene un tamaño prefijado que no se puede modificar, excepto por el método **rehash**. Inicialmente las posiciones no usadas contienen **null**.
  - **count** contiene el número de copias de un elemento dado que almacena la tabla . Así, si el entero almacenado en la posición **i** de **count** es un 3 significa que conjunto contiene tres copias del elemento almacenado en la posición **i** de **table**.
  - Si la posición **i** de **count** contiene un 0 indicará que la posición **i** fue usada pero el elemento fue borrado. Si contiene un **-1**, indicará que esa posición en **table** nunca ha sido usada. Obviamente al iniciarse el vector **count** todas sus celdas contendrán un **-1**.
  - **rehash()** Dado el elemento **item** calcula su posición dentro de la tabla usando el resultado de una función de dispersión y adaptado a las dimensiones de la tabla.
  - **size** almacena el número de elementos del conjunto. **used** almacena el número de entradas de la tabla que han sido usadas.
- (a) **(0.5 puntos)** Implementa el método privado **int hash(E item)** que dado el valor de **item** devuelva un entero con el resultado de la función de dispersión adaptada a las dimensiones de la tabla.
- (b) **(1 punto)** Implementa un método público **int count(E item)** que devuelva el número de copias de **item** que contenga el conjunto. Devolverá 0 si no contiene ninguna.
- (c) **(1.5 puntos)** Implementa el método público **boolean add(E item)** que añade una copia del elemento al conjunto. Si es la primera que añade devolverá **true**, y si ya existían más copias **false**.

- 3.2** (2 puntos) Deseamos escribir una clase genérica **ChainedHashSet<T>** que implemente la interfaz **Set<T>** usando una tabla de dispersión. Las colisiones se resolverán usando el método del encadenamiento, pero con una peculiaridad: los elementos se incluirán en una única cadena no circular de nodos doblemente enlazados, tal y como se puede ver en la figura.



En el ejemplo de la figura, el elemento que corresponde a la posición 1 es el 8, a la posición 3, el 17, 3, y el 45, y así con el resto de posiciones. Las referencias **first** y **last** indican el primer y último nodo de la cadena, respectivamente. Esta construcción permite recorrer eficientemente los elementos del conjunto en ambas direcciones con un iterador.

La parte privada de la clase tiene esta definición:

```
public class ChainedHashSet<T> implements Set<T>{
    private class Node {
        public T data;
        public Node next = null;
        public Node prev = null;

        public Node(T item) {data = item;}
    }

    private Node[] table;
    private int size;
    private Node first = null;
    private Node last = null;

    private int hash(T item) {...}
    private boolean equalsNull(T item1, T item2) {...}
    ...
}
```

La clase permite almacenar elementos **null**. El método **hash** calcula la posición correspondiente a un elemento dentro de la tabla. **equalsNull** compara si dos elementos son iguales, teniendo en cuenta que pueden ser **null**.

Implementa el método **remove** que tome un elemento, lo busque en la estructura, y si lo encuentra lo borre. Devolverá **true** si lo encuentra, o **false** en caso contrario. El método deberá ser eficiente y respetar la estructura descrita en los párrafos anteriores.

```
public boolean remove(T item)
```

**3.3 (2.5 puntos)** Deseamos implementar un variante del tipo de datos conjunto que admite elementos repetidos. Para ello decidimos usar una tabla de dispersión para almacenar los elementos. Las colisiones se resolverán mediante el método de direccionamiento abierto. Tendrá la peculiaridad de que por cada elemento almacenado en la tabla se guardará una cuenta con el número de copias que aparece en el conjunto. De tal manera que la primera vez que se guarda un elemento en el conjunto el número de la cuenta será 1. Si vuelve a guardarse un segunda copia del mismo elemento la cuenta se incrementará a 2.

La parte privada de la clase será:

```
public class HashBag<E> implements Set<E> {
    private ArrayList<E> table;
    private ArrayList<Integer> count;
    private int size;
    private int used;

    private int hash(E item);
    ...
}
```

- **table** almacena los datos de la tabla de dispersión. Tiene un tamaño prefijado que no se puede modificar, excepto por el método **rehash**. Inicialmente las posiciones no usadas contienen **null**.
- **count** contiene el número de copias de un elemento dado que almacena la **table**. Así, si el entero almacenado en la posición **i** de **count** es un 3 significa que conjunto contiene tres copias del elemento almacenado en la posición **i** de **table**.
- Si la posición **i** de **count** contiene un 0 indicará que la posición **i** fue usada pero el elemento fue borrado. Si contiene un **-1**, indicará que esa posición en **table** nunca ha sido usada. Obviamente al iniciarse el vector **count** todas sus celdas contendrán un **-1**.
- **hash(E item)** Calcula la posición del elemento usando una función de dispersión. El resultado será un entero dentro del rango [0 ... **table.size()**-1]
- **size** almacena el número de elementos del conjunto.
- **used** almacena el número de entradas de la tabla que han sido usadas.

(a) **(0.5 puntos)** Implementa un método constructor que tome como parámetro el tamaño de la tabla.

```
public HashBag<E>(int tam)
```

(b) **(2.0 puntos)** Implementa el método **rehash**

```
public void rehash()
```

Deberá calcular el factor de carga de la tabla, y si el número de casillas usadas es mayor que la mitad de la tabla procederá a doblar el tamaño de la ésta y recolocar todos los elementos.

No se proporciona ningún otro método. Si crees que necesitas alguno para resolver el ejercicio deberás implementarlo.

*Curso 2015/16 Examen Final 2<sup>a</sup> Convocatoria*

**Solución:**

```

public HashBag(int tam) {
    // Crea las tablas
    this.table = new ArrayList<E>();
    this.count = new ArrayList<Integer>();

    // Las inicializa con los valores adecuados
    for (int i=0; i < tam; i++) {
        table.add(null);
        count.add(-1);
    }
    this.size = 0;
    this.used = 0;
}

private void rehash() {
    if ((used/table.size()) < 0.5)
        return;

    // Guarda las viejas tablas
    List<E> tableAux = table;
    List<Integer> countAux = count;

    // Crea nuevas tablas con el doble de tamaño
    table = new ArrayList<E>();
    count = new ArrayList<Integer>();

    // Inicializa las nuevas tablas con los valores adecuados
    for (int i = 0; i < tableAux.size()*2; i++) {
        table.add(null);
        count.add(-1);
    }
    used = 0;

    // Reinserta todos los elementos usando la función de dispersión.
    for(int i = 0; i < tableAux.size(); i++) {
        if (countAux.get(i) >0) {
            int pos = hash(tableAux.get(i));

            while (count.get(pos)!=-1)
                pos = (pos + 1) % table.size();

            table.set(pos, tableAux.get(i));
            count.set(pos, countAux.get(i));
            used++;
        }
    }
}

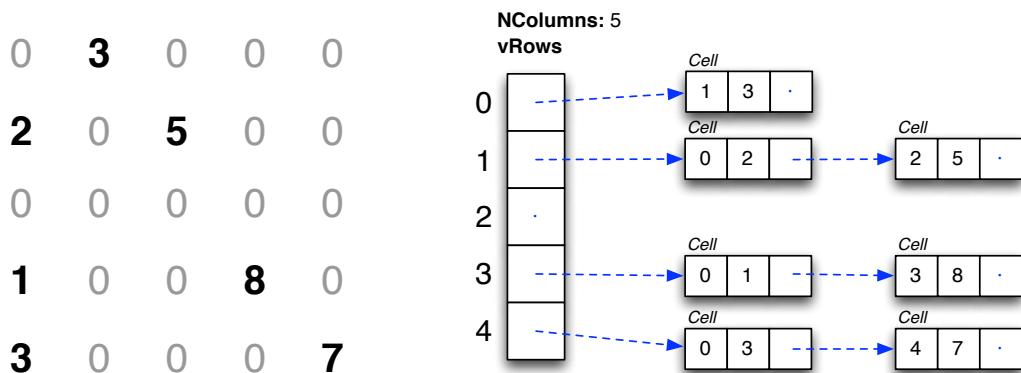
```

**3.4 (3 puntos)** En matemáticas, una matriz dispersa es una matriz de dos dimensiones tal que la mayoría de los elementos son cero. Si quisieramos representar de forma eficiente una matriz dispersa, la mejor forma sería almacenar sólo aquellos elementos que no son nulos. Para ello implementamos en la clase **SparseMatrix** con esta parte privada:

```
public class SparseMatrix {
    private class Cell {
        public int column;
        public double value;
        public Cell next = null;
    }
    private int NColumns = 0;
    private Cell[] vRows = null;
    ...
}
```

- **Cell**: Representa una celda de la matriz. Almacena el número de columna de la celda, su valor (distinto de cero) y un enlace a la siguiente celda de la misma fila.
- **NColumns** almacenan el número de columnas de la matriz.
- **vRows** Vector de filas. Cuando una fila contenga algún elemento no nulo, almacenará una referencia a la primera celda no nula. En caso de no haber elementos distintos de cero almacenará **null**.

Gráficamente, la matriz de la izquierda tendría este aspecto:



Las celdas se numeran empezando desde la fila 0 y la columna 0.

Implementa un método estático de la clase **SparseMatrix** que sume una matriz dispersa con la almacenada en el objeto actual. El resultado se almacenará sobre el objeto actual.

```
public void Add(SparseMatrix matrix)
```

En el caso de que alguna de las dimensiones de **matrix** sea mayor que la actual se incrementarán las dimensiones de esta última.

*Curso 2014/15 Examen Final 1<sup>a</sup> Convocatoria*

**Solución:**

```

public void add(SparseMatrix matrix) {
    // Comparamos el número de filas de las matrices
    if (vRows.length < matrix.vRows.length) {
        // Es necesario ampliar el tamaño de vRows.
        Cell [] aux = vRows;
        vRows = new Cell[matrix.vRows.length];
        for (int i = 0; i < aux.length; i++)
            vRows[i] = aux[i];
    }

    // Comparamos el número de columnas de la matriz
    if (NColumns < matrix.NColumns)
        NColumns = matrix.NColumns;

    // Se suman ambas matrices fila a fila
    for (int i = 0; i < matrix.vRows.length; i++) {
        Cell c1 = vRows[i]; // Recorrerá las celdas de vRows[i]
        Cell prev = null; // Guardará la última celda visitada en vRows[i]
        Cell c2 = matrix.vRows[i]; // Recorrerá las celdas de matrix.vRows[i]

        while (c1 != null && c2 != null) {
            // Mientras haya celdas en ambas filas
            if (c1.column < c2.column) {
                // Se avanza
                prev = c1;
                c1 = c1.next;
            } else if (c1.column == c2.column) {
                // Se deben sumar ambas celdas
                c1.value += c2.value;
                c2 = c2.next;
                if (c1.value == 0.0) {
                    // Si la suma es cero esa celda debe eliminarse de vRows[i]
                    if (prev == null)
                        vRows[i] = c1.next;
                    else
                        prev.next = c1.next;
                    c1 = c1.next;
                } else
                    prev = prev.next;
            } else { // c1.column > c2.column
                // Debe copiarse c2 antes de c1
                Cell n = new Cell();
                n.column = c2.column;
                n.value = c2.value;
                n.next = c1;
                prev.next = n;
                prev = n;
                c2 = c2.next;
            }
        }

        while (c2 != null) {
            // Hemos terminado vRows[i] pero aun quedan
            // elementos en matrix.vRows[i]. Se copian
            Cell n = new Cell();
            n.value = c2.value;

```

```
    n.column = c2.column;
    if (prev == null)
        vRows[i] = n;
    else
        prev.next = n;
    prev = n;
}
}
}
```

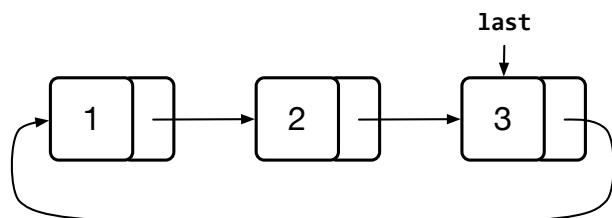
**3.5** (2 puntos) Disponemos de la clase genérica `SingleLinkedList <T>` que implementa la interface `List <T>` usando una cadena de nodos simplemente enlazados de forma circular. Se ha implementado de modo que los elementos con valor `null` no se añaden a la lista.

La parte privada de la clase contiene estas declaraciones.

```
public class SingleLinkedList<T> implements List<T> {
    private class Node {
        public T data;
        public Node next;
    }

    private Node last = null;
    private int size = 0;
```

Dónde `last` hace referencia al nodo que contiene el último elemento de la lista, y `size` almacena el número de nodos de la lista. Por ejemplo la lista `[1, 2, 3]` tendrá, gráficamente, este aspecto



Implementa el método `reverse` que invierte el orden de la lista, de forma que los primeros sean los últimos y los últimos los primeros. Su definición es:

```
public void reverse()
```

Algunos ejemplos de su efecto:

- `this : [a, b, c, d, e, f, g]; reverse() → this : [g, f, e, d, c, b, a]`
- `this : [a, b]; reverse() → this : [b, a]`
- `this : [a]; reverse() → this : [a]`
- `this : [ ]; reverse() → this : []`

Nota: En la implementación de este método no puedes usar ninguno de los métodos de la interface `List` sino que debes resolverlo mediante la modificación de enlaces entre nodos.

*Curso 2017/18 Examen Final 2<sup>a</sup> Convocatoria*

**Solución:**

Dos posibles versiones de `reverse`:

```
public void reverse() {  
    if (size <= 1) return;  
    last = last.next;  
    if (size==2) return;  
  
    Node prev = last;  
    Node current = prev.next;  
    Node aux = current.next;  
  
    while (aux.next != current) {  
        current.next = prev;  
        prev = current;  
        current = aux;  
        aux = aux.next;  
    }  
}  
  
public void reverse2() {  
    if (size <= 1) return;  
    last = last.next;  
    if (size==2) return;  
  
    Node prev = last;  
    Node current = prev.next;  
    Node aux = current.next;  
  
    for (int i = 0; i < size; i++) {  
        current.next = prev;  
        prev = current;  
        current = aux;  
        aux = aux.next;  
    }  
}
```

**3.6 (1 punto)** La clase **HashCollection<T>** implementa la interfaz **Collection** usando una tabla de dispersión con resolución de colisiones mediante direccionamiento abierto. La clase admite elementos repetidos que se almacenarán en distintas posiciones de la tabla, y no admite el elemento **null** como un elemento válido en la colección.

La parte privada de la clase es:

```
public class HashCollection<E> implements Collection<E> {
    private ArrayList<E> table;
    private ArrayList<Boolean> used;
    private int size;
    private int dirty;

    private int hash(E item);
    ...
}
```

- **table** almacena los datos de la tabla de dispersión. Tiene un tamaño prefijado que no se puede modificar, excepto por el método **rehash**. Inicialmente las posiciones no usadas contienen **null**. También contienen **null** si ha sido usadas pero están actualmente vacías.
- **used** indica si la posición de la tabla ha sido usada en alguna ocasión: **true** si ya ha sido usada o **false** si no lo ha sido nunca.
- **hash(E item)** Calcula la posición del elemento usando una función de dispersión. El resultado será un entero dentro del rango **[0 ... table.size() - 1]**
- **size** almacena el número de elementos del conjunto.
- **dirty** almacena el número de entradas de la tabla que han sido usadas.

Añade un método **compress** a la clase que para todos los elementos repetidos elimine todas las copias menos una. Es decir, si el conjunto contiene los elementos **{f, g, h, g, a, b, a, a, c}** al terminar deberá contener los elementos **{f, g, h, a, b, c}**. El método devolverá el número de elementos eliminados.

```
public int compress()
```

*Curso 2018/19 Examen Final 1<sup>a</sup> Convocatoria*

**Solución:**

```
public int compress() {
    Set<T> in = new HashSet<>();
    int count = 0;
    for (int i = 0; i < table.size(); i++) {
        T current = table.get(i);
        if (current != null) {
            if (in.contains(current)) {
                table.set(i, null);
                count++;
                size--;
            } else
                in.add(current);
        }
    }
    return count;
}
```

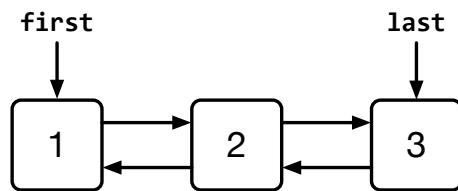
**3.7** (2 puntos) Disponemos de la clase genérica `DoubleLinkedList<T>` que implementa la interface `List<T>` usando una cadena no circular de nodos doblemente enlazados. Se ha implementado de manera que los elementos `null` no se añaden a la cadena.

La parte privada de la clase es:

```
public class EDDoubleLinkedList<T> implements List<T> {
    private class Node {
        public T data;
        public Node next;
        public Node prev;
    }

    private Node first = null;
    private Node last = null;
    private int size = 0;
```

Dónde `first` hace referencia al nodo que contiene el primer elemento de la lista y `last` al nodo que contiene el último. `size` almacena el número de nodos de la lista. Por ejemplo la lista [1, 2, 3] tendrá, gráficamente, este aspecto:



Implementa el método `duplicate` que tomará todos los elementos de la lista y los añadirá a la lista original intercalados entre los originales pero en orden inverso.

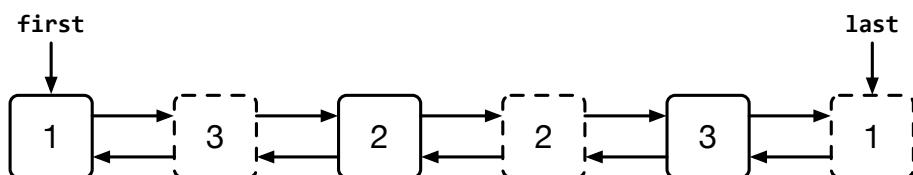
```
public void duplicate()
```

Nota: En la implementación de este método no puedes usar ninguno de los métodos de la interface `List` sino que debes resolverlo mediante la modificación de enlaces entre nodos.

Algunos ejemplos de su efecto (subrayados los elementos nuevos):

- this: [a, b, c, d]; `duplicate()` → this: [a, d, b, c, c, b, d, a]
- this: [f, g, h]; `duplicate()` → this: [f, h, g, g, h, f]
- this: [x, y]; `duplicate()` → this: [x, y, y, x]
- this: [a]; `duplicate()` → this: [a, a]
- this: [ ]; `duplicate()` → this: []

Así quedaría el ejemplo gráfico tras aplicar la operación `duplicate`. Los nodos nuevos aparecen con línea discontinua.



**Solución:** Dos posibles soluciones

```
public void duplicate() {
    if (size == 0) return;

    int pos = 0;

    Node forward = first;
    Node backward = last;

    while (forward != null) {
        Node n = new Node();
        n.next = backward.next;
        n.prev = backward;
        n.data = forward.data;

        if (backward == last)
            last = n;
        else
            backward.next.prev = n;

        backward.next = n;
        backward = backward.prev;

        backward = backward.prev;
        forward = forward.next;
        if (pos > (size/2))
            forward = forward.next;
        pos++;
    }
    size = size * 2;
}

public void duplicate() {
    Node forward = first;

    while (forward != null) {
        Node n = new Node();
        n.prev = forward;
        n.next = forward.next;
        forward.next = n;
        if (forward != last)
            n.next.prev = n;
        else
            last = n;
        forward = forward.next;
    }

    Node backward = last;
    forward = first;

    while (forward != null) {
        backward.data = forward.data;
        forward = forward.next.next;
        backward = backward.prev.prev;
    }
    size = size * 2;
}
```

**3.8** (2,5 puntos) Tenemos la clase genérica **EDHashSet<T>** que implementa todos los métodos de la interfaz **Set<T>** excepto **iterator**. Para ello usa una tabla de dispersión con resolución de colisiones mediante encadenamiento. Parte de su implementación es esta:

```
public class EDHashSet<T> implements Set<T> {
    class Node {
        public T data;
        public Node next = null;
        public Node(T elem) { data = elem; }
    }

    protected Node[] table;
    protected int size = 0;

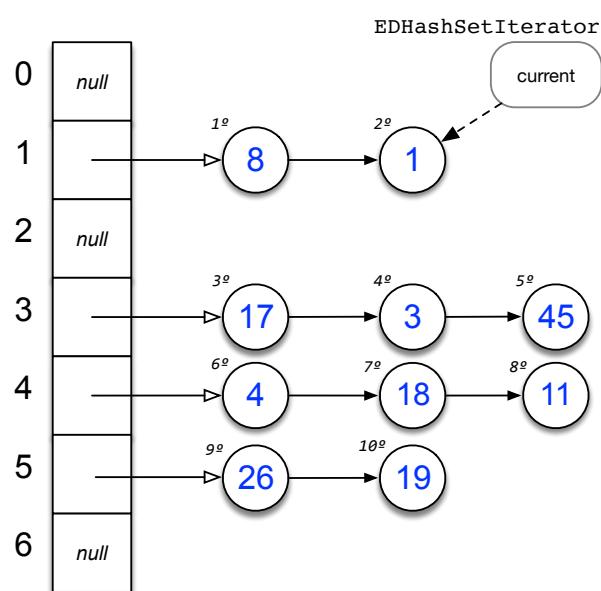
    int hash(T item) { ... }
```

El método **hash** calcula la posición de el elemento **item** dentro de **table** usando una función de dispersión.

En este ejercicio vas a completar la implementación de la clase **EDHashSetIterator** que implementa la interfaz **Iterator<T>**. Se trata de un clase interna de **EDHashSet** que implementa un iterador para esta última. La clase se define como:

```
private class EDHashSetIterator implements Iterator<T> {
    Node current = null;
}
```

La clase continua una referencia **current** a un nodo. Esta referencia va a ir pasando por todos nodos de la tabla a medida que se llame al método **next**. El funcionamiento de la clase se puede observar en la siguiente figura:



El atributo **current** almacena un enlace al nodo que contiene el siguiente dato que devolverá el iterador. En el caso de la figura cuando se use el método **next()** éste devolverá el valor 1 y **current** pasará a hacer referencia al siguiente nodo, que, en este caso, es el que almacena el 17.

Los nodos se recorren en el orden exemplificado en la figura, es decir, desde el principio de la tabla hacia el final, y en cada cadena, del primer al último nodo.

El método **iterator** de la clase **EDHashSet<T>** es:

```
public Iterator<T> iterator() {
    return new EDHashSetIterator();
}
```

- (a) (1 punto) Implementa un método privado **findNext** de la clase **EDHashSetIterator**, tal que tome un enlace a un nodo **n** de la tabla de dispersión y devuelva el nodo con el siguiente dato según el orden explicado anteriormente.

```
private Node findNext(Node n)
```

Si el parámetro es **null** devolverá el primer nodo disponible. Si no hay siguiente nodo el resultado será **null**.

- (b) (1,5 puntos) Implementa el constructor por defecto de la clase **EDHashSetIterator** y los métodos **next** y **hasNext**. El constructor inicializará la clase para que haga referencia al primer nodo, y los otros dos métodos se comportarán de acuerdo a lo especificado en la interfaz **Iterator<T>**.

**NOTA:** recuerda que el método **next** devuelve la excepción **NoSuchElementException** cuando no hay siguiente elemento.

*Curso 2019/20 Examen Final 1<sup>a</sup> Convocatoria*

**3.9** (2 puntos) La clase `EDDoubleLinkedList<T>` implementa la interfaz `List<T>` usando una cadena de nodos doblemente enlazados no circular y sin nodo cabecera. La clase no admite elementos con valor `null`.

```
public class EDDoubleLinkedList<T> implements List<T> {
    private class Node {
        public T data;
        public Node next = null;
        public Node prev = null;

        public Node(T item) { data = item; }
    }
    private Node head = null;
    private Node tail = null;
    private int size = 0;
    ...
}
```

Implementa el método `trim`, que tome dos elementos de la lista, `start` y `end` y borre los elementos anteriores a `start` y posteriores a `end`, preservando los que se encuentren entre ellos, ambos incluidos

```
public boolean trim(T start, T end)
```

En el caso de que alguno de los parámetros aparezca repetido en la lista, se tomará como límite la primera aparición de `start` y/o la última de `end`. En el caso de que `start` aparezca más tarde que `end` se preservarán los elementos situados entre `end` y `start`, ambos incluidos.

El método devolverá `true` si se ha eliminado algún elemento de la lista; `false` en caso contrario, si algún de los parámetros es `null` o si alguno de ellos no está incluido en la lista.

Implementad el método modificando solo los enlaces entre nodos, es decir, sin usar ningún otro método de la interfaz `List`.

jemplos:

- `this : [2, 3, 5, 4, 8]; trim(3, 4) → true, this : [3, 5, 4]`
- `this : [2, 3, 4, 3, 8, 4, 2]; trim(3, 4) → true, this : [3, 4, 3, 8, 4]`
- `this : [2, 2, 4, 3, 5, 4, 8]; trim(3, 4) → true, this : [3, 5, 4]`
- `this : [2, 4, 5, 3, 2, 0]; trim(3, 4) → true, this : [4, 5, 3]`
- `this : [3, 5, 4]; trim(3, 4) → false, this : [3, 5, 4]`
- `this : [2, 3, 5, 4, 8]; trim(3, 3) → true, this : [3]`
- `this : [2, 3, 5, 3, 8]; trim(3, 3) → true, this : [3, 5, 3]`

*Curso 2018/19 Examen Final 1<sup>a</sup> Convocatoria*

**3.10 (2 puntos)** La clase genérica **ChainedHashSet<T>** implementa la interfaz **Set<T>** mediante una tabla de dispersión con resolución de colisiones mediante encadenamiento. La clase admite la inclusión de **null** como un elemento más del conjunto. La parte privada de la clase incluye las siguientes definiciones:

```
public class ChainedHashSet<T> implements Set<T> {
    private class Node {
        public T data;
        public Node next;
    }

    private Node table[];
    private int size = 0;

    private int hash(Object o) {
        if (o == null) return 0;
        return (o.hashCode() & Integer.MAX_VALUE) % table.length;
    }
}
```

Completa la implementación de la clase con el método **remove** de la interfaz **Set**. Nota: En la implementación de este método no puedes usar ninguno de los métodos de la interfaz **Set** sino que debes resolverlo mediante la modificación de enlaces entre nodos.

```
public boolean remove(Object o)
```

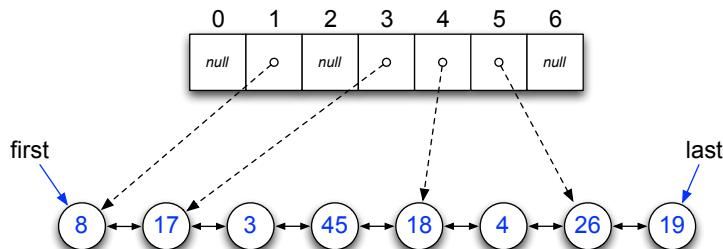
*Curso 2017/18 Examen Final 2<sup>a</sup> Convocatoria*

**3.11 (1 punto)** Una tabla de dispersión de tamaño inicial 6 puede almacenar enteros positivos (incluyendo el 0) y resuelve las colisiones mediante direccionamiento abierto. Como función de dispersión utiliza  $h(i) = (i \% T)$  donde  $i$  es el valor que se desea almacenar y  $T$  el tamaño de la tabla. Las posiciones libres se marcan almacenando un -1 y las borradas un -2. Se produce un *rehashing* cuando las casillas usadas son al menos 5/6 del total de casillas la tabla. En el *rehashing* se dobla el tamaño de la tabla.

Muestra gráficamente el contenido de cada casilla de la tabla tras realizar cada una de estas operaciones: insertar 0, 2, 4, y 10, borrar 4, insertar 8, borrar 2 y 10, e insertar 15.

*Curso 2018/19 Examen Final 2<sup>a</sup> Convocatoria*

**3.12 (2 puntos)** Deseamos escribir una clase genérica **ChainedHashBag<T>** que implemente la interfaz **Collection <T>**. Esta clase permitirá almacenar, buscar y borrar elementos de un mismo tipo pudiendo haber elementos repetidos y sin mantener ningún tipo de orden. Para su implementación decidimos usar una tabla de dispersión. Las colisiones se resolverán usando el método del encadenamiento con una peculiaridad: los elementos se incluirán en una única cadena no circular de nodos doblemente enlazados tal y como puede verse en la figura.



En el ejemplo de la figura, el elemento que corresponde a la posición 1 es el 8, que aparece repetido, a la posición 3, el 17, 3, y el 45, y de igual forma con el resto de posiciones. Las referencias **first** y **last** indican el primer y último nodo de la cadena, respectivamente. Esta construcción permite a un iterador recorrer eficientemente los elementos del conjunto en ambas direcciones.

La parte privada de la clase tiene esta definición:

```
public class ChainedHashBag<T> implements Set<T>{
    private class Node {
        public T data;
        public Node next = null;
        public Node prev = null;

        public Node(T item) {data = item;}
    }

    private Node[] table;
    private int size;
    private Node first = null;
    private Node last = null;

    private int hash(T item) {...}
    private void rehash() {...}
    ...
}
```

El método **hash** calcula la posición correspondiente a un elemento dentro de la tabla. **rehash** comprueba la ocupación de la tabla y realiza una redispersión cuando es necesario.

Implementa el método **add** que tome un elemento y lo añada a la tabla en la posición que le corresponda. La tabla no admite elementos con valor **null** por lo que en esos casos se devolverá **false**. El método deberá ser eficiente y respetar la estructura descrita en los párrafos anteriores.

```
public boolean add(T item)
```

**3.13 (2 puntos)** Disponemos de la clase genérica `DoubleLinkedList<T>` que implementa la interfaz genérica `List<T>` usando una cadena de nodos doblemente enlazados no circular sin nodo cabecera.

```
public class DoubleLinkedList<T> implements List<T> {
    private class Node {
        public T data;
        public Node next = null;
        public Node prev = null;
    }
    private Node head = null;
    private Node tail = null;
    private int size = 0;

    public void DoubleLinkedList() {...}
    ...
}
```

Añade un método `trim` a la clase para borrar elementos del principio y del final de la lista. Recibirá como parámetro un entero  $i$  y borrará los  $i$  primeros y los  $i$  últimos elementos de la lista. El método devolverá una lista nueva que contendrá todos los elementos borrados en el mismo orden en el que se encontraban en la lista inicial. La lista devuelta estará vacía si  $i$  es igual o menor que 0. Implementad el método modificando solo los enlaces entre nodos, es decir, sin usar ningún otro método de la interfaz List. La cabecera será:

```
public List<T> trim(int i)
```

Ejemplos:

- `this : [a, b, c, d, e, f, g]; trim(1) → this : [b, c, d, e, f], return : [a, g]`
- `this : [a, b, c, d, e, f, g]; trim(3) → this : [d], return : [a, b, c, e, f, g]`
- `this : [a, b, c, d, e, f, g]; trim(4) → this : [], return : [a, b, c, d, e, f, g]`
- `this : [a, b, c, d, e, f, g]; trim(0) → this : [a, b, c, d, e, f, g], return : []`

*Curso 2016/17 Examen Final 2<sup>a</sup> Convocatoria*

**3.14** (2 puntos) Sea la clase **DoubleLinkedList** que implementa la interfaz **List** mediante una cadena no circular de nodos doblemente enlazados. Añade un método **removeSublist** que borre todos los elementos dados entre dos índices, **index1** $\leq$  **index2**, ambos incluidos.

A continuación tienes algunos ejemplos del funcionamiento del método

```
[ a, b, c, d, e, f, g, h] removeSubList(3, 5) => [ a, b, c, g, h]
[ a, b, c, d, e, f, g, h] removeSubList(4, 4) => [ a, b, c, d, f, g, h]
[ a, b, c, d, e, f, g, h] removeSubList(5, 9) => [ a, b, c, d, e ]
[ a, b, c, d, e, f, g, h] removeSubList(-2,5) => [ g, h]
[ a, b, c, d, e, f, g, h] removeSubList(5, 3) => [ a, b, c, d, e, f, g, h]
```

```
public class DoubleLinkedList<E> implements List<E> {
    private class Node {
        E data;
        Node next;
        Node prev;
    }
    Node head;
    Node tail;
    int size;

    public void removeSubList(int index1, int index2);
}
```

*Curso 2015/16 Examen Final 2<sup>a</sup> Convocatoria*

**3.15** (2 puntos) Dada una lista, vamos a implementar una función **flip**. Ésta coge los elementos de la lista de dos en dos y los intercambia de posición. Ejemplo:

[1, 2, 3, 4, 5] -> [2, 1, 4, 3, 5]

Supón que tenemos la clase **EDLinkedList<T>** que implementa la interfaz **List<T>**. Esta clase usa una lista de nodos simplemente enlazados de forma circular sin nodo cabecera. Implementa **flip** como un nuevo método de la clase.

Ésta es la parte privada de la clase:

```
public class EDLinkedList<T> implements List<T> {

    private static class Node<T> {
        private T data;
        private Node<T> next;

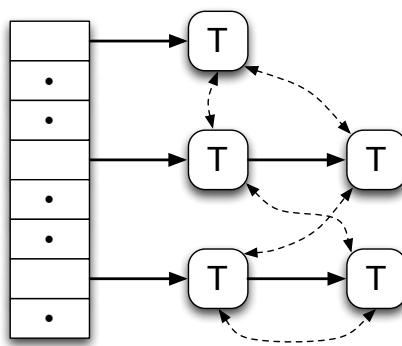
        public Node(T element) {
            data = element;
            next = null;
        }
    }

    private Node<T> first = null;
    private int size = 0;
    ...
}
```

*Curso 2013/14 Examen Final 2<sup>a</sup> Convocatoria*

**3.16 (3 puntos)** Pretendemos implementar una variante de conjunto que preserve el orden de inserción de los elementos, de forma que al recorrer los elementos con un iterador estos sean devueltos en el orden de inserción. Para implementar esta solución nos planteamos implementar el conjunto genérico usando una tabla de dispersión con resolución de colisiones mediante encadenamiento. A su vez los nodos se enlazarán en una lista doblemente enlazada circular. La nueva clase `LinkedHashSet<T>` implementará las interfaces `Set` y `List`.

El concepto sería descrito por este gráfico



- (a) (1 punto) Escribe la definición de la clase y las estructuras internas necesarias para almacenar los datos de la manera descrita. Escribe también el constructor.
- (b) (2 puntos) Implementa el método `boolean add(T item)` asumiendo que no se permitirá insertar elementos repetidos. Ten en cuenta que este método debe funcionar como el `add` de conjuntos y listas a la vez.

*Curso 2013/14 Examen Final 1<sup>a</sup> Convocatoria*

**3.17 (2,5 puntos)** Un polinomio  $P(x)$  de grado  $n \geq 0$  de una sola variable  $x$  es un sumatorio de monomios del la forma

$$P(x) = \sum_{i=0}^n a_i x^i$$

donde  $a_i, i = 0 \dots n$  son números reales. Cada uno de los sumandos  $a_i x^i$  se denomina *monomio*, donde  $a_i$  es el coeficiente e  $i$  es un entero que indica el grado del monomio.

Vas a implementar una clase para manipular polinomios de una variable. Por eficiencia, la clase sólo almacenará los monomios cuyo coeficiente  $a_i$  sea distinto de cero.

1. Define la parte privada de la clase que almacenará los datos.
2. Implementa un constructor que tome como entrada un vector `float [v]` con los coeficientes del polinomio. El elemento `v[i]` almacenará el coeficiente  $a_i$ .
3. Incluye un método que tome como parámetro otro polinomio y lo sume al almacenado en el objeto.

*Curso 2011/12 Examen Final 2<sup>a</sup> Convocatoria*

**3.18 (3 puntos)** Vamos a implementar una tabla de dispersión con resolución de colisiones mediante encadenamiento, con la peculiaridad de que se permitirán elementos con la misma clave. Para ello definimos la siguiente clase genérica:

```
public class ChainHashTable<K, V> {
    private class Node {
        public K key;
        public V value;
        public int count=1;

        // Constructor de un nodo
        Node(K key, V value) {...}
    }

    private int REHASH_THRESHOLD;
    private List<Node>[] table;
    private int size;
    private int unique;

    private void rehash();

    ChainHashTable (int initial_size);
    public int addEntry(K key, V value);
    public int removeEntry(K key);
    public K getEntry(K key);
    public int getCount(K key);
    ...
}
```

- **K** es el tipo de la clave y **V** el de los valores asociados.
- La clase interna **Node** almacena un par clave/valor y en **count** el número de veces que se ha insertado un par con esa clave. El constructor del nodo inicializa la cuenta a 1.
- **table** contiene las listas de cada posición de la tabla de dispersión. Si en una posición de la tabla no se han insertado elementos, la posición correspondiente tiene el valor **null**.
- **unique** contiene el número de elementos almacenados en la tabla sin tener en cuenta los repetidos, y **size** el número de elementos total en la teniendo en cuenta los repetidos.
- Se realiza *rehashing* cuando el valor de **unique** es mayor que el de **REHASH\_THRESHOLD**. El tamaño de **table** se dobla y el valor **REHASH\_THRESHOLD** se multiplica por dos.
- **addEntry** añade un par clave/valor nuevo. En caso de que ya existiera un par con en la misma clave, reemplaza el valor. Devuelve el número de elementos que se han insertado con esa misma clave.
- **removeEntry** elimina una ocurrencia del par clave/valor. Devuelve cero si sólo había un par con esa clave, o no había ninguno.

Implementa los métodos **addEntry** y **rehash**.

*Curso 2012/13 Examen Final 2<sup>a</sup> Convocatoria*

**3.19 (2.5 puntos)** Tenemos una clase genérica `DoubleLinkedList<T>` que implementa completamente la interfaz `List<T>` mediante nodos doblemente enlazados circularmente. La definición de su parte privada es la siguiente:

```
public class DoubleLinkedList<T> implements List<T> {
    private static class Node<T> {
        public T data;
        public Node<T> next;
        public Node<T> prev;
        ...
    }
    private Node<T> head = null;
    private int size = 0;
    ...
}
```

Añade un método a esta clase para mezclar listas:

```
public void shuffle(List<T> lista)
```

El método tomará una lista `lista` y la mezclará con lista local `this`, modificándola. De tal forma que el primer elemento de `lista` se insertará entre el primero y el segundo de la lista local, el segundo elemento de `lista` entre el segundo y el tercero de la lista local, y así sucesivamente. Si la lista local fuese demasiado corta se insertarían los elementos sobrante de `lista` al final de la lista local.

Ejemplos:

- `lista : [2, 4, 5]; this : [9, 8, 7, 6] → this : [9, 2, 8, 4, 7, 5, 6]`
- `lista : [2, 4, 5]; this : [99, 100] → this : [99, 2, 100, 4, 5]`

*Curso 2012/13 Examen Final 1<sup>a</sup> Convocatoria*

**3.20 (1 punto)** ¿ Cual es el propósito de la función de dispersión en las tablas de dispersión? ¿Qué condiciones ha de cumplir una buena función de dispersión?

*Curso 2011/12 Examen Final 2<sup>a</sup> Convocatoria*

**3.21 (2.5 puntos)** Deseamos implementar el tipo de datos **HashBag**, una colección desordenada que admite elementos repetidos. Para ello decidimos usar una tabla de dispersión para almacenar los elementos. Las colisiones se resolverán mediante el método de direccionamiento abierto. Si un elemento es añadido varias veces, ocupará varios huecos dentro de la tabla de dispersión.

```
public class HashBag<T> implements Collection<T> {
    private ArrayList<T> table = new ArrayList<>(100);
    private ArrayList<Boolean> used = new ArrayList<>(100);
    private int size = 0;
    private int dirty = 0;
    private int rehashThreshold;

    private int hash(Object o) {...}
```

- **table** almacena los datos de la tabla de dispersión. Tiene un tamaño prefijado que no se puede modificar, excepto por el método **rehash**. Inicialmente las posiciones no usadas contienen **null**. Al borrar un elemento se guardará **null** en la posición de **table** que ocupaba.
- **used** indica si una posición de la tabla está siendo usada o lo ha sido en el pasado.
- **size** almacena el número de elementos del conjunto.
- **dirty** almacena el número de entradas de la tabla que han sido usadas.
- **rehashThreshold** indica el umbral a partir del cual se realiza el *rehashing*.

- (a) (1 punto) Implementa el método **remove** que borra todas las copias de un elemento **T o** en el **HashBag**. Devolverá **true** si ha borrado algún elementos o **false** si no. En el caso de que el parámetro sea **null** lanzará la excepción **NullPointerException**.

```
public boolean remove(T o)
```

- (b) (1.5 puntos) Implementa el método **rehash** que amplia el tamaño de la tabla y redispersiona todos los elementos. La condición para realizar la operación es **dirty > rehashThreshold**

```
private void rehash()
```

Deberás resolver el apartado sin usar otros métodos de la interfaz **Collection**.

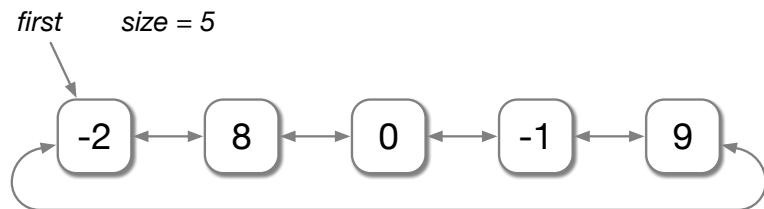
**3.22 (2 puntos)** Disponemos de la clase genérica `DoubleLinkedList<T>` que implementa la interface `List<T>` usando una cadena circular de nodos doblemente enlazados. Se ha implementado de manera que los elementos `null` no se añaden a la lista.

La parte privada de la clase es:

```
public class DoubleLinkedList<T> implements List<T> {
    private class Node {
        public T data;
        public Node next;
        public Node prev;
    }

    private Node first = null;
    private int size = 0;
```

El atributo `first` hace referencia al nodo que contiene el primer elemento de la lista, `size` almacena el número de nodos de la lista. Por ejemplo la lista `[-2, 8, 0, -1, 9]` tendrá, gráficamente, este aspecto:



Implementa un método de la clase que reordene los elementos de la lista de forma que quede separada en dos mitades. En la primera mitad quedarán los elementos cuyo `hashCode` sea negativo, y en la segunda mitad aquellos cuyo `hashCode` sea 0 o mayor. Dentro de cada mitad los elementos conservarán el orden inicial entre ellos.

```
public void separate()
```

Nota: En la implementación de este método no puedes usar ninguno de los métodos de la interface `List` sino que debes resolverlo mediante la modificación de enlaces entre nodos.

A continuación puedes ver algunos ejemplos del resultado de la aplicación del método sobre listas de enteros (ten en cuenta para un `Integer` su `hashCode()` devuelve el mismo valor entero):

- this: [-2, 8, 0, -1, 9]; `separate()` → this: [-2, -1, 8, 0, 9]
- this: [ 3, 5, 1, 0, -1 ]; `separate()` → this: [ -1, 3, 5, 1, 0 ]
- this: [ -3, -5, -1 ]; `separate()` → this: [ -3, -5, -1 ]
- this: [ 3, 5, 1 ]; `separate()` → this: [ 3, 5, 1 ]
- this: [ 2, -2 ]; `separate()` → this: [ -2, 2 ]
- this: [ ]; `separate()` → this: [ ]