EJERCICIOS RESUELTOS DE SMALLTALK

Ejercicio N° 1

i. Se le solicita que defina en Smalltalk una clase Matriz (para representar este concepto), la cual deberá ser una subclase de OrderedCollection. Describa claramente que modelo emplea para representar el concepto de matriz. Se le solicita que defina los métodos básicos para su creación, inspección y modificación:
new: *n*
at: *unPar*
at: *unPar* put: *valor*
Para la representación del objeto *unPar* puede emplear la clase Point.
ii. Con las especificaciones del punto **i**. defina el método **simétrica**, el cual deberá retornar *true*, si la matriz receptora es una matriz simétrica. Recuerde que la definición de matriz simétrica es la siguiente:

$$A = A^T$$

Defina todos los métodos que considere necesario para definir **simétrica**.

Resolución:
La matriz se representará como una lista de filas, utilizando el comportamiento heredado de la superclase, donde cada fila será un Array (cuyo tamaño es equivalente al número de columnas).

Clase Matriz

```
OrderedCollection subclass: #Matriz
  instanceVariableNames:
    'filas columnas '
  classVariableNames: ''
  poolDictionaries: ''
```

Métodos de clase Matriz

```
new: unEntero
"crea y responde una matriz cuadrada de unEntero*unEntero"
^self new: unEntero por: unEntero.

new: rows por: cols
      "crea y responde una matriz de rows*cols"
|ret|
ret := super new: rows.
ret  inicializar: (rows @ cols).
^ret.
```

Métodos de instancia Matriz

```
at: unPar
"responde el valor contenido en (unPar x, unPar y)"
|row col|
row := unPar x.
col := unPar y.
^((super at: row) at: col).

at: unPar put: unValor
"almacena unValor en la posicion (unPar x, unPar y)
 y responde el valor almacenado"
|row col|
row := unPar x.
col := unPar y.
^((super at: row) at: col put: unValor).

columnas
"responde el numero de columnas que posee la Matriz"
^columnas.
```

esCuadrada
"responde true si la Matriz tiene igual numero de filas que de columnas"
^(filas = columnas).

esSimetrica
"responde true si la Matriz es simetrica"
|ret cuentaFila cuentaCol|

```
(self esCuadrada) ifFalse: [ ^false ].
cuentaFila := 1.
cuentaCol := 1.
ret := self transpuesta.
filas timesRepeat: [ columnas timesRepeat: [ ( (self at: cuentaFila @ cuentaCol) =
                                                (ret at: cuentaFila @ cuentaCol) )
                                                    ifFalse: [ ^false ].
                                                    cuentaCol := cuentaCol + 1 ].
                                cuentaCol := 1.
                                cuentaFila := cuentaFila +1  ].
^true.
```

filas
"responde el numero de filas que posee la Matriz"
^filas.

inicializar: *unPar*
"inicializa el contenido de la Matriz, así como el número de filas y de columnas"

```
filas:= unPar x.
columnas:= unPar y.

1 to: filas do: [ :cuentaFila |
                        self add: (Array new: columnas) .
                        1 to: columnas do: [ :cuentaCol |
                                (super at: cuentaFila) at: cuentaCol put: 0 ].
                ].
```

transpuesta
"responde la transpuesta de una Matriz"
|ret|

```
ret := Matriz new:  self columnas por: self filas.
1 to: filas do: [ :cuentaFila | 1 to: columnas do:
                        [ :cuentaCol | ret at: cuentaCol @ cuentaFila put: (self at: cuentaFila @ cuentaCol)
                        ]
                ].
^ret.
```

Ejercicio N° 2

Se le solicita que cree una subclase de Array, en la cual se le solicita que defina los siguientes métodos que permiten que mantenga la historia de los valores asignados y la posibilidad de fijar valores máximos y mínimos para cada posición del arreglo:
historia: *i* ; retorna una colección con todos los valores que fueron asignados a la posición *i*.
invertir ; revierte el orden de los valores del arreglo receptor del mensaje, se modifica el estado del mismo.
at: *i* valorMax: *n* ; asignar un valor máximo *n* para la posición *i*.
at: *i* valorMin: *n* ; asignar un valor mínimo *n* para la posición *i*.

Describa la estructura de la subclase, es decir los atributos y demás métodos que sean necesarios.

Resolución:

Clase ArrayConHistoria
```
Array variableSubclass: #ArrayConHistoria
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

Métodos de clase ArrayConHistoria

new
```
"Si no se especifica el tamaño del ArrayConHistoria, crea y responde  uno de 12 elementos"
^self new: 12.
```

new: *unNumero*
```
"Crea y responde un ArrayConHistoria de tamaño 'unNumero' "
^(super new: unNumero) inicializar: unNumero.
```

Métodos de instancia ArrayConHistoria

at: *unaPosicion*
```
"responde el valor almacenado en la posición 'unaPosicion' "
^(super at: unaPosicion) getValor.
```

at: *unaPosicion* ***put:*** *unValor*
```
"coloca 'unValor' en la posicion 'unaPosicion'.
  ^(super at: unaPosicion) setValor: unValor
```

at*: i* ***valorMax****: n*
```
^(super at: i) valorMax: n.
```

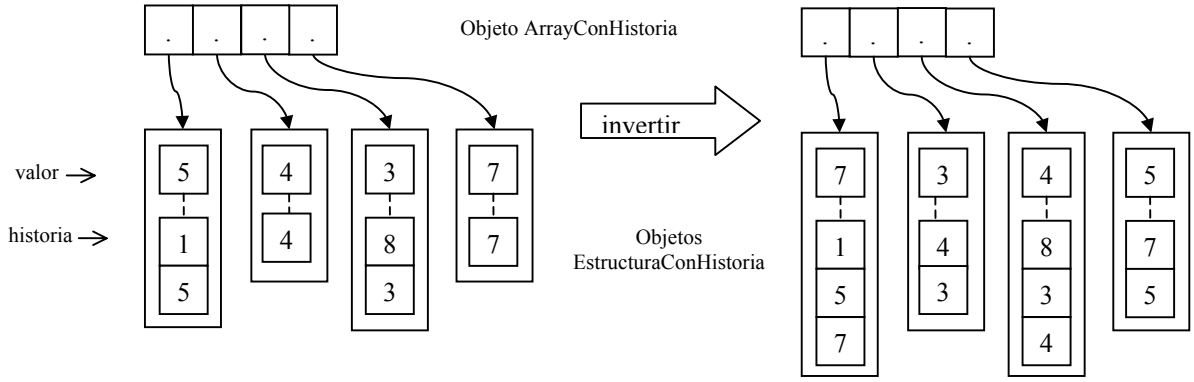
at*: i* ***valorMin****: n*
```
^(super at: i) valorMin: n.
```

historia: *unaPosicion*
```
"responde una OrderedCollection que almacena la historia
de los valores asignados en 'unaPosicion' "
  ^(super at: unaPosicion ) getHistoria
```

inicializar: *unValor*
```
"inicializa el ArrayConHistoria"
1 to: unValor do: [ :param | super at: param put: (EstructuraConHistoria new) ]
```

invertir
"invierte el orden de los elementos actuales en el ArrayConHistoria. "
|avance regreso temp|
avance := 1.
regreso := self size.
[avance < regreso] whileTrue: [
temp := self at: avance.
self at: avance put: (self at: regreso).
self at: regreso put: temp.
avance := avance + 1.
regreso := regreso - 1
].

Ejemplo de Inversión de un objeto ArrayConHistoria



<u>Clase EstructuraConHistoria</u>

Object subclass: #EstructuraConHistoria
instanceVariableNames:
'valor historia valorMax valorMin '
classVariableNames: "
poolDictionaries: "

Métodos de clase EstructuraConHistoria

new
^super new inicializar.

Métodos de Instancia de EstructuraConHistoria

getHistoria
^historia.

getValor
^valor.

inicializar
historia := OrderedCollection new.

```
setValor: unValor
"almacena un valor en la estructura, si esta permitido por las cotas de valorMin y valorMax
 y responde el valor almacenado"

valorMin notNil ifTrue: [ valorMin > unValor
                        ifTrue: [ self error: 'El valor es menor a la cota inferior.' ]
                    ].
valorMax notNil ifTrue: [ valorMax < unValor
                        ifTrue: [ self error: 'El valor es mayor a la cota superior.' ]
                    ].

valor  := unValor.
historia add: unValor.
^unValor


valorMax: unValor
valorMax := unValor.

valorMin: unValor
valorMin := unValor.
```

Ejercicio N° 3

Se desea construir la estructura básica de un editor de texto tomando como clases básicas: documento, página, línea, palabra y caracter. En esta definición elemental, una página tendrá un número máximo de líneas y una línea tendrá un número máximo de caracteres. Una palabra tendrá un dado número de caracteres.

Documento
new
Crea un objeto documento vacío.
insertarPágina
Crea una página vacía al final del documento.

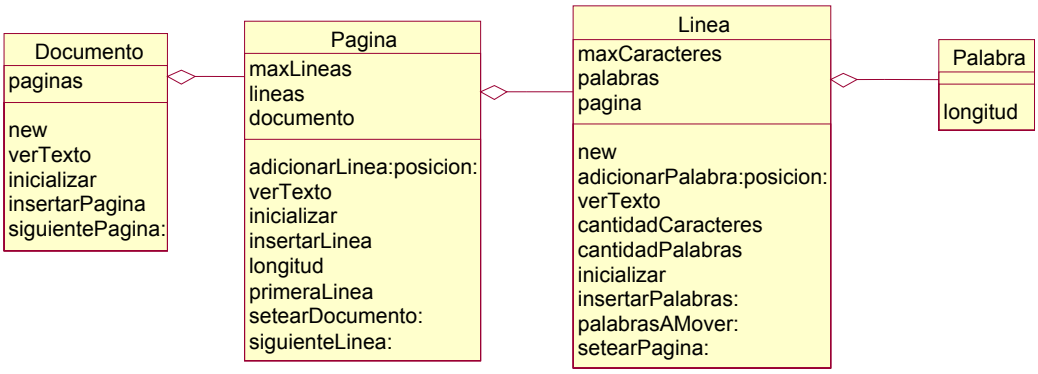
Página
new
Crea un objeto página, el cual no contiene ninguna línea.
adicionarLínea: *línea* **posición**: *n*
Incorpora una línea en una posición dada de la página. Si como consecuencia de la incorporación de la línea se excede el número máximo de líneas de la página, la última línea deberá ser enviada a la próxima página, y así sucesivamente. La línea incorporada puede ser vacía.

Línea
new
palabrasAMover: *palabra*
Retorna una OrderedColection con las palabras que quedarían fuera de la línea como consecuencia de incorporar *palabra* al comienzo de la misma. Si la incorporación no produce que se exceda el número de caracteres máximo de la línea, el método retorna una colección vacía.
adicionarPalabra: *palabra* **posición:** *n*
Incorpora un objeto palabra en la línea, como la palabra enésima. Si la palabra, en esa posición excede la longitud máxima de la línea, el método anuncia la imposibilidad de incorporarla en esa posición. Si la puede incorporar, pero algunas de las palabras posteriores ya no tienen más espacio, deberán ser detectadas y enviadas al comienzo de la línea siguiente.

Palabra
longitud
retorna la longitud de la palabra

Se le solicita que defina las clases y métodos explicitados (y todos aquellos que considere convenientes), teniendo especial atención en identificar las relaciones entre las distintas clases.

Resolución:
Diagrama de clases indicando las relaciones de composición, atributos y métodos propuestos:



Clase Documento

```
Object subclass: #Documento
  instanceVariableNames:
    'paginas '
  classVariableNames: ''
  poolDictionaries: ''
```

Métodos de clase Documento

new

```
  ^super new inicializar.
```

Métodos de Instancia Documento

verTexto

```
"Devuelve un string que contiene el texto del documento"
    | cr cadena |
    cr := (Character value: 13) asString.
    cadena := ''.
    1 to: paginas size do: [ :i |
            cadena := cadena , '[Page ' , i asString , ']' , cr , (paginas at: i) verTexto, cr  ].

    ^cadena.
```

inicializar

```
   paginas := OrderedCollection new.
```

insertarPagina

```
    | nuevaPagina |
    nuevaPagina := Pagina new.
    nuevaPagina setearDocumento: self.
    paginas add: nuevaPagina.
    ^nuevaPagina.
```

siguientePagina: *unaPagina*

```
  "Devuelve la página que se encuentra a continuación de la pasada como argumento,
    si es la última página agrega una nueva"

   paginas last == unaPagina ifTrue: [ ^self insertarPagina ].
   ^paginas after: unaPagina ifNone: [ self error: 'La página solicitada no se encuentra en el documento.' ].
```

Clase Pagina

```
Object subclass: #Pagina
  instanceVariableNames:
    'maxLineas lineas documento '
  classVariableNames: ''
  poolDictionaries: ''
```

Métodos de clase Página

new

```
  ^super new inicializar.
```

Métodos de instancia Pagina

adicionarLinea: *unaLinea* ***posicion:*** *n*

```
  " Incorpora una línea en una posición dada de la página. "
   | siguientePagina |

   (n >= 1) & (n <= (self longitud + 1))
       ifTrue: [
                  lineas add: unaLinea beforeIndex: n.
                  unaLinea setearPagina: self.

                  " Si como consecuencia de la incorporación de la línea se excede
                    el número máximo de líneas de la página, la última línea deberá
                    ser enviada a la próxima página, y así sucesivamente. "

                 self longitud > maxLineas
                       ifTrue:  [
                                    siguientePagina := documento siguientePagina: self.
                                    siguientePagina adicionarLinea: lineas last posicion: 1.
                                    lineas removeLast.
                                ].

                  ^unaLinea
               ]
       ifFalse: [  self error: 'La posición indicada para la linea no es válida en la página.'  ].
```

verTexto

```
   | cr cadena |
   cr := (Character value: 13) asString.

   lineas isEmpty ifTrue: [ ^'' ].
   cadena := (lineas at: 1) verTexto.
   2 to: lineas size do: [ :i | cadena := cadena , cr , (lineas at: i) verTexto ].
   ^cadena.
```

inicializar

```
  lineas := OrderedCollection new.
  maxLineas := 3.
```

insertarLinea

```
   | nuevaLinea |
   nuevaLinea := Linea new.
   self adicionarLinea: nuevaLinea posicion: (self longitud + 1).
   ^nuevaLinea.
```

longitud

```
  " Devuelve el número de líneas del receptor "
  ^lineas size.
```

```
primeraLinea
    " Devuelve la primera línea de la página "
    ^lineas first.

setearDocumento: unDocumento
   documento := unDocumento.

siguienteLinea: unaLinea
  " Devuelve la linea que se encuentra a continuación de la pasada como argumento,
    si es la última línea agrega una nueva "
   | siguientePagina |
   lineas last == unaLinea ifTrue: [
                        self longitud = maxLineas
                           ifTrue: [   siguientePagina := documento siguientePagina: self.
                                       siguientePagina longitud = 0 ifTrue: [ siguientePagina insertarLinea ].
                                      ^siguientePagina primeraLinea
                                   ]
                              ifFalse: [  ^self insertarLinea   ]
                                       ].

   ^lineas after: unaLinea ifNone: [ self error: 'La linea solicitada no se encuentra en la página.' ].
```

Clase Linea

```
Object subclass: #Linea
  instanceVariableNames:
    'maxCaracteres palabras pagina '
  classVariableNames: ''
  poolDictionaries: ''
```

Métodos de clase Linea

```
new
   ^super new inicializar.
```

Métodos de instancia Linea

```
adicionarPalabra: unaPalabra posicion: n
    " Incorpora una palabra en una posición dada de la línea. "
    | buffer siguienteLinea |
    (n >= 1) & (n <= (self cantidadPalabras + 1))
         ifTrue: [
                   buffer := self palabrasAMover: unaPalabra.
                   n <= (self cantidadPalabras + 1 - buffer size)
                         ifTrue:  [
                                    palabras add: unaPalabra beforeIndex: n.

                                    buffer notEmpty ifTrue: [
                                       buffer do: [ :p | palabras removeLast ].
                                       siguienteLinea := pagina siguienteLinea: self.
                                       siguienteLinea insertarPalabras: buffer
                                    ].
                                ]
                         ifFalse:  [  self error: 'La palabra no puede ser adicionada en la posición solicitada.'  ].
                   ^unaPalabra
                ]
         ifFalse: [  self error: 'La posición indicada para la palabra no es válida en la línea.'  ].
```

```
verTexto
    | cadena |
    palabras isEmpty ifTrue: [ ^'' ].
    cadena := palabras at: 1.
    2 to: palabras size do: [ :i | cadena := cadena , ' ' , (palabras at: i) ].
    ^cadena.

cantidadCaracteres
  ^self verTexto size.

cantidadPalabras
  ^palabras size.

inicializar
   palabras := OrderedCollection new.
   maxCaracteres := 20.


insertarPalabras: listaPalabras
    " Insertar una lista de palabras al comienzo de la linea, si el número de caracteres
      es mayor al máximo, enviar las palabras sobrantes a la linea siguiente "

    | buffer siguienteLinea |
    palabras addAllFirst: listaPalabras.
    buffer := OrderedCollection new.
    [ self cantidadCaracteres > maxCaracteres ]
          whileTrue:  [   buffer addFirst: palabras last.
                          palabras removeLast     ].

    buffer notEmpty ifTrue: [
         siguienteLinea := pagina siguienteLinea: self.
         siguienteLinea insertarPalabras: buffer
                                ].

palabrasAMover: palabra
    | indice cuenta |
    cuenta := palabra size.
    cuenta > maxCaracteres
        ifTrue: [ self error: 'La palabra no entra en la línea.' ].

    palabras isEmpty ifTrue: [ ^OrderedCollection new ].

    indice := 1.
    [ (indice <= palabras size) and: [ cuenta := cuenta + (palabras at: indice) size + 1.
                                       cuenta <= maxCaracteres ]
    ] whileTrue:
              [ indice := indice + 1 ].

    ^palabras copyFrom: indice to: palabras size.


setearPagina: unaPagina
   pagina := unaPagina.
```

Aclaración: De acuerdo a la estrategia empleada para la resolución de este ejercicio, no se definió una nueva clase *Palabra* sino que se utilizó directamente la clase *String* provista por Smalltalk y el método *size* de la misma para hallar la longitud.

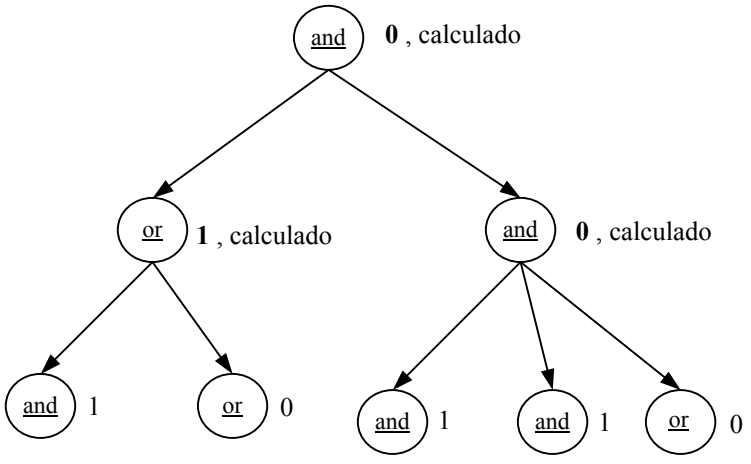
Casos de prueba para la solución propuesta:

```
| doc p1 lin1 lin2 |
    doc := Documento new.
    p1 := doc insertarPagina.
    lin1 := p1 insertarLinea.
    lin2 := p1 insertarLinea.
    lin1 adicionarPalabra: 'pepe' posicion: 1 ;
        adicionarPalabra: 'es' posicion: 2;
        adicionarPalabra: 'objetos' posicion: 3;
        adicionarPalabra: 'en' posicion: 3.
    lin2 adicionarPalabra: 'juan' posicion: 1 ;
        adicionarPalabra: 'sabe' posicion: 2;
        adicionarPalabra: 'smalltalk' posicion: 3;
        adicionarPalabra: 'en' posicion: 3;
        adicionarPalabra: 'programar' posicion: 3.
    doc verTexto.

    lin1 adicionarPalabra: 'experto' posicion: 3.
    doc verTexto.
```

Ejercicio N° 4

Se pretende crear las clases necesarias para mantener un árbol AndOr. Este grafo se caracteriza por que los arcos son dirigidos, cada nodo tiene un único padre y a cada nodo puede tener asociado un valor 0 o 1. Otra característica es que hay dos tipos de nodos, nodos tipo AND y nodos tipo OR. En el caso de los nodos AND, tendrán asociado un valor 1, si todos sus hijos tienen asociados valores 1. En el caso de los nodos OR, tendrán asociado un valor 1, si alguno de sus hijos tiene asociado un valor 1.

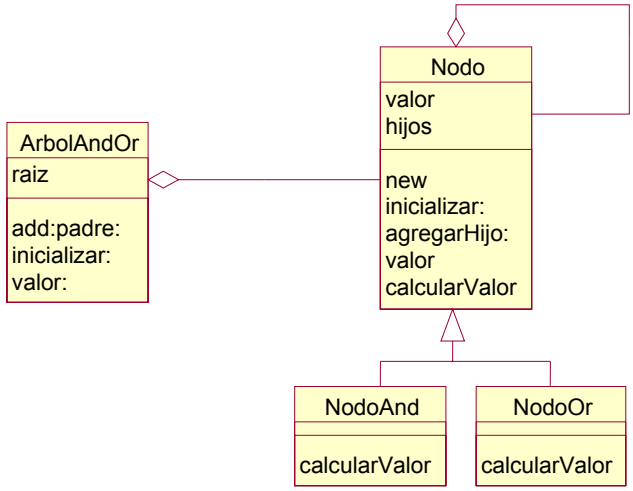


A los nodos hojas (nodos sin hijos) se le asignará un valor 1 o 0. En el caso de los restantes nodos, el valor asociado se calculará a partir del valor asociado a sus hijos. Se le solicita que defina las clases necesarias, especificando claramente las variables instancia y los métodos que sean necesarios para mantener el árbol. Además deberá definir los siguientes métodos:

Arbol:
add: *nodo* **padre**: *nodoPadre* ; adiciona un *nodo* al árbol receptor y se incorpora como hijo de *nodoPadre* (los argumentos son objetos AND o OR, no strings)
valor: *nodo* ; calcula el valor asociado *nodo*

Además de la clase Arbol, defina todas las clases que considere necesarias, así como los métodos de creación e inserción de nodos al árbol

Resolución:



Clase ArbolAndOr

Object subclass: #ArbolAndOr
instanceVariableNames:
'raiz '
classVariableNames: ''
poolDictionaries: ''

Métodos de clase ArbolAndOr

new: *unaRaiz*
^super new inicializar: unaRaiz.

Métodos de Instancia ArbolAndOr

add: *nodoHijo* ***padre:*** *nodoPadre*
((nodoHijo isKindOf: Nodo) & (nodoPadre isKindOf: Nodo))
ifTrue: [
nodoPadre agregarHijo: nodoHijo
]
ifFalse: [self error: 'Los argumentos no son válidos'].

inicializar: *unaRaiz*
raiz := unaRaiz.

valor: *nodo*
^nodo valor

Clase Nodo *"Es una clase abstracta para los distintos tipos de nodos: AND y OR"*
Object subclass: #Nodo
instanceVariableNames:
' valor hijos '
classVariableNames: ''
poolDictionaries: ''

Métodos de Clase Nodo

new: *unValor*
^super new inicializar: unValor.

Métodos de Instancia Nodo

inicializar: *unValor*
valor := unValor.
hijos := OrderedCollection new.

agregarHijo: *unNodo*
hijos add: unNodo.

valor
hijos isEmpty ifTrue: [^valor]
ifFalse: [^self calcularValor].

calcularValor
^self implementedBySubclass

Clase NodoAnd

Nodo subclass: #NodoAnd
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''

Métodos de Instancia NodoAnd

calcularValor
|valores|
valores := hijos collect: [:n | n valor].
(valores includes: 0) ifTrue: [^0] ifFalse: [^1].

Clase NodoOr
Nodo subclass: #NodoOr
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''

Métodos de Instancia NodoOr

calcularValor
|valores|
valores := hijos collect: [:n | n valor].
(valores includes: 1) ifTrue: [^1] ifFalse: [^0].

Casos de Prueba para la solución propuesta.

```
| a b c d e f g h i j k l m n o p raiz arbol |

a := NodoAnd new: 1.
b := NodoAnd new: 1.
c := NodoOr new: 1.
d := NodoAnd new: 1.
e := NodoOr new: 0.
f := NodoAnd new: 1.
g := NodoOr new: 0.
h := NodoOr new: 1.
i := NodoAnd new: 1.
j := NodoAnd new: 0.
k := NodoOr new: 0.
l := NodoOr new: 1.
m := NodoAnd new: 1.
n := NodoAnd new: 1.
o := NodoOr new: 0.
p := NodoOr new: 1.
raiz := NodoAnd new: 0.

arbol := ArbolAndOr new: raiz.

arbol add: f padre: raiz.
arbol add: g padre: raiz.
arbol add: h padre: raiz.
arbol add: m padre: raiz.
arbol add: i padre: h.
arbol add: j padre: i.
arbol add: l padre: m.
arbol add: k padre: l.
arbol add: e padre: f.
arbol add: d padre: e.
arbol add: c padre: d.
arbol add: b padre: d.
arbol add: a padre: b.
arbol add: n padre: e.
arbol add: o padre: n.
arbol add: p padre: n.

arbol valor: i.
```