Ejercicio 1

- Escriba los algoritmos para los siguientes procs y calcule su complejidad
  - `proc agregarAtras`(inout l : ListaEnlazada$\langle T \rangle$, in $t : T$)
  - `proc obtener`(in l : ListaEnlazada$\langle T \rangle$, in $i : \mathbb{Z}$) $: T$
  - `proc eliminar`(inout l : ListaEnlazada$\langle T \rangle$, in $i : \mathbb{Z}$)
  - `proc concatenar`(inout $l1$: ListaEnlazada$\langle T \rangle$, in $l2$: ListaEnlazada$\langle T \rangle$)
- Escriba el invariante de representación para este módulo en castellano

- Dado el siguiente invariante de representación, indique si es correcto. En caso de no serlo, corrijalo:

  `pred InvRep` (l: ListaEnlazada$\langle T \rangle$) {
  $accesible(l.primero, l.ultimo) \land largoOK(l.primero, l.longitud)$
  }
  `pred largoOK` (n: NodoLista$\langle T \rangle$, $largo$: $\mathbb{Z}$) {
  $(n = null \land largo = 0) \lor (largoOK(n.siguiente, largo - 1))$
  }
  `pred accesible` (n_0: NodoLista$\langle T \rangle$, n_1: NodoLista$\langle T \rangle$) {
  $n_1 = n_0 \lor (n_0.siguiente \neq null \land_L accesible(n_0.siguiente, n_1))$
  }

Figura 1: Enunciado Problema 1

`NodoLista<T>` = Struct⟨`valor: T, siguiente: NodoLista<T>`⟩

```
Modulo ListaEnlazada<T> implements Secuencia<T> {
    var primero: NodoLista<T>
    var ultimo: NodoLista<T>
    var longitud: int

    pred listaDe (nodo: NodoLista<T>, s: seq⟨T⟩) {
        (|s| = 0 ⟶ nodo = Null) ∧
        (|s| > 0 ⟶ (nodo ≠ Null ∧L nodo.valor = head(s) ∧L listDe(nodo.siguiente, tail(s))))
    }
    pred abs (l: ListaEnlazada<T>, l': Seceuencia<T>) {
        |l'.s| = l.longitud ∧L listaDe(l.primero, l'.s)
    }
    pred invRep (l: ListaEnlazada<T>) {
        l.longitud ≥ 0 ∧
        (l.longitud = 0 ⟶ (l.primero = Null ∧ l.ultimo = Null)) ∧
        (l.longitud = 1 ⟶ (l.primero ≠ null ∧ l.primero = l.ultimo)) ∧
        (l.longitud > 1 ⟶ (l.primero ≠ Null ∧ l.ultimo ≠ Null ∧ l.primero ≠ l.ultimo)) ∧L
        sinCiclos(l.primero, {}) ∧L largoOK(l.primero, l.longitud) ∧ alcanzable(l.primero, l.ultimo)
    }
    pred sinCiclos (nodo: NodoLista<T>, visitados: conj⟨NodoLista<T>⟩) {
        nodo = Null ∨L (nodo ∉ visitados ∧L sinCiclos(nodo.siguiente, visitados ∪ {nodo}))
    }
    pred largoOK (nodo: NodoLista<T>, longitud: ℤ) {
        (longitud = 0 ⟶ nodo = null) ∧
        (longitud ≠ 0 ⟶ nodo ≠ null ∧L largoOK(nodo.siguiente, longitud − 1))
    }
    pred alcanzable (n0: NodoLista<T>, n1: NodoLista<T>) {
        (n0 = n1) ∨ (n0 ≠ Null ∧L alcanzable(n0.siguiente, n1))
    }
```

`proc` agregarAtras (inout l: `ListaEnlazada<T>`, in t: T)

```
var nodo: NodoLista<T>
    nodo:= new NodoLista<T>()
    nodo.valor:= t
    nodo.siguiente:= null

if l.primero == null && l.ultimo == null
  l.primero:= nodo
else
  l.ultimo.siguiente:= nodo
endif

l.ultimo:= nodo
l.longitud:= l.longitud + 1

return
```

`proc` obtener (in l: `ListaEnlazada<T>`, in i: int) : T

```
var nodoActual: NodoLista<T>
    nodoActual:= l.primero

while (nodoActual != null && i > 0) do
  nodoActual:= nodoActual.siguiente
  i:= i - 1
endwhile

return nodoActual.valor
```

`proc` eliminar (inout l: `ListaEnlazada<T>`, in i: int)

```
if i == 0 then
  l.primero:= l.primero.siguiente
  l.ultimo:= l.longitud > 1 ? l.ultimo : null
else
  var nodoActual: NodoLista<T>
      nodoActual:= l.primero

  while (i > 1) do
    nodoActual:= nodoActual.siguiente
    i:= i - 1
  endwhile

  nodoActual.siguiente:= nodoActual.siguiente.siguiente
  l.ultimo:= nodoActual.siguiente != null ? l.ultimo : nodoActual
endif

l.longitud:= l.longitud - 1

return
```

`proc` concatenar (inout lista1: `ListaEnlazada<T>`, in lista2: `ListaEnlazada<T>`)

```
var nodoActual: NodoLista<T>
    nodoActual:= lista2.primero

while (nodoActual != null) do
  agregarAtras(lista1, nodoActual.valor)
  nodoActual:= nodoActual.siguiente
endwhile

return
```

}

Ejercicio 2. Implemente el TAD `ConjuntoAcotado<T>` (definido en el apunte de TADs) usando la siguiente estructura.

```
Módulo ConjuntoArr<T> implementa ConjuntoAcotado<T> {
    var datos: Array<T>
    var tamaño: int
}
```

- Escriba el invariante de representación y la función de abstracción.
- Escriba los algoritmos para las operaciones **conjVacío** y **pertence**
- Escriba el algoritmo para la operación **agregar**
- Escriba los algoritmos para las operaciones **unir** e **intersecar**.
- Escriba el algoritmo para la operación **sacar**.
- Calcule la complejidad de cada una de estas operaciones
- Qué cambios haría en su implementación si se quiere que la operación **agregar** sea lo más rápida posible? Y si se quiere acelerar la operación **buscar**? Indique los cambios en la estructura, el invariante de representación, la función de abstracción y los algoritmos.

Figura 2: Enunciado Problema 2

Modulo ConjuntoAcotadoArr<T> implements ConjuntoAcotado<T> {

var **datos**: $Array\langle T\rangle$

var **tamaño**: int

pred sinRepetidos (c: ConjuntoAcotadoArr<T>,) {

$(\forall i, j : \mathbb{Z})\ (0 \leq i, j < c.tamaño \land i \neq j \longrightarrow_L c.datos[j] \neq c.datos[i])$

}

pred abs (c: ConjuntoAcotadoArr<T>, c': ConjuntoAcotado<T>) {

$c'.cota = length(c.datos) \land |c'.elems| = c.tamaño \land (\forall t : T)\ (t \in c'.elems \leftrightarrow t \in subseq(c.datos, 0, c.tamaño))$

}

pred invRep (c: ConjuntoAcotadoArr<T>) {

$0 \leq c.tamaño \leq length(c.datos) \land sinRepetidos(c) \land (\forall j : \mathbb{Z})\ (0 \leq j < c.tamaño \longrightarrow_L def(c.datos[j]))$

}

proc conjVacío (in cota: int) : ConjuntoAcotadoArr<T>

```
var datos: Array<T>
    datos:= new Array<T>(cota)

res.datos:= datos
res.tamaño:= 0

return res
```

proc pertenece (in c: ConjuntoAcotadoArr<T>, in t: T) : Boolean

```
var i: int
    i:= 0

while (i < c.tamaño) do
  if c.datos[i] == t then
    return true
  endif

  i:= i + 1
endwhile

return false
```

```
% agregarSiNoPertenece: For the sake of declaratividad %
```

proc agregarSiNoPertenece (inout c: ConjuntoAcotadoArr<T>, in t: T)

```
if !pertenece(c, t) then
  c.datos[c.tamaño]:= t
  c.tamaño:= c.tamaño + 1
endif

return
```

proc agregar (inout c: ConjuntoAcotadoArr<T>, in t: T)

```
agregarSiNoPertenece(c, t)

return
```

proc sacarPorIndice (inout c: ConjuntoAcotadoArr<T>, in i: int)

```
c.datos[i]:= c.datos[c.tamaño - 1]
c.tamaño:= c.tamaño - 1

return
```

proc sacar (inout c: ConjuntoAcotadoArr<T>, in t: T)

```
var i: int
    i:= 0

while (i < c.tamaño) do
  if c.datos[i] == t then
    sacarPorIndice(c, i)

    return
  endif

  i:= i + 1
endwhile

return
```

proc unir (inout c1: ConjuntoAcotadoArr<T>, in c2: ConjuntoAcotadoArr<T>)

```
var i: int
    i:= 0

while (i < c2.tamaño) do
  agregarSiNoPertenece(c1, c2.datos[i])
  i:= i + 1
endwhile

return
```

proc intersecar (inout c1: ConjuntoAcotadoArr<T>, in c2: ConjuntoAcotadoArr<T>)

```
var i: int
    i:= 0

while (i < c1.tamaño) do                // O(n)
  if !pertenece(c2, c1.datos[i]) then // O(m)
    sacarPorIndice(c1, i)             // O(1)
  endif

  i:= i + 1
endwhile

return
```

}

Ejercicio 3. Implementar el TAD Conjunto$\langle T \rangle$ (definido en el apunte de TADs) usando la siguiente estructura

```
Módulo ConjuntoLista<T> implementa Conjunto⟨T⟩ {
    var datos: ListaEnlazada<T>
    var tamaño: int
}
```

- Escriba el invariante de representación y la función de abstracción.
- Escriba los algoritmos para las operaciones **conjVacío** y **pertence**
- Escriba el algoritmo para la operación **agregar**, **agregarRápido** y **sacar**
- Escriba los algoritmos para las operaciones **unir** e **intersecar**.
- Calcule la complejidad de cada una de estas operaciones

Figura 3: Enunciado Problema 3

Modulo ConjuntoLista<T> implements Conjunto<T> {
    var **datos**: ListaEnlazada<T>
    var **tamaño**: *int*

    pred sinRepetidos (c: ConjuntoLista<T>) {
        $(\forall i, j : \mathbb{Z})\ ((0 \leq i, j < |c.datos.s| \wedge_L c.datos.s[i] = c.datos.s[j]) \longrightarrow i = j)$
    }
    pred abs (c: ConjuntoLista<T>, c': Conjunto<T>) {
        $|c'.elems| = c.tamaño \wedge (\forall t : T)\ (t \in c'.elems \leftrightarrow t \in c.datos.s)$
    }
    pred invRep (c: ConjuntoLista<T>) {
        $c.tamaño = |c.datos.s| \wedge sinRepetidos(c)$
    }

    proc conjVacio () : ConjuntoLista<T>

```
var nuevaListaEnlazada: ListaEnlazada<T>
    nuevaListaEnlazada:= new listaEnlazadaVacia()

res.datos:= nuevaListaEnlazada
res.tamaño:= 0

return res
```

proc buscarIndicePorValor (in c: ConjuntoLista<T>, in t: T) : Boolean

```
// Devuelve -1 si el elemento no pertenece
// Devuelve 0 <= i < c.tamaño si el elemento pertenece
var i: int
    i:= 0
var indice: int
    indice:= -1

while (i < c.datos.tamaño() && indice == -1) do
  if c.datos.obtener(i) == t then
    indice:= i
  endif

  i:= i + 1
endwhile

return indice
```

proc pertenece (in c: ConjuntoLista<T>, in t: T) : Boolean

```
var indice: int
    indice:= buscarIndicePorValor(c, t)

return indice != -1
```

proc agregarRapido (inout c: ConjuntoLista<T>, in t: T)

```
c.datos.agregarAtras(t)
c.tamaño = c.datos.tamaño()

return
```

```
% agregarSiNoPertenece: For the sake of declaratividad %
```

proc agregarSiNoPertenece (inout c: ConjuntoLista<T>, in t: T)

```
if !pertenece(c, t) then
  agregarRapido(c, t)
endif

return
```

proc agregar (inout c: ConjuntoLista<T>, in t: T)

```
agregarSiNoPertenece(c, t)

return
```

    proc sacar (inout c: ConjuntoLista<T>, in t: T)

```
if pertenece(c, t) then
  var indice: int
      indice:= buscarIndicePorValor(c, t)

  c.datos.eliminar(indice)
  c.tamaño = c.datos.tamaño()
endif

return
```

    proc unir (inout c1: ConjuntoLista<T>, inout c2: ConjuntoLista<T>)

```
var i: int
    i:= 0

while (i < c2.tamaño()) do
  agregarSiNoPertenece(c1, c2.datos.obtener(i))
  i:= i + 1
endwhile
```

    proc intersecar (inout c1: ConjuntoLista<T>, in c2: ConjuntoLista<T>)

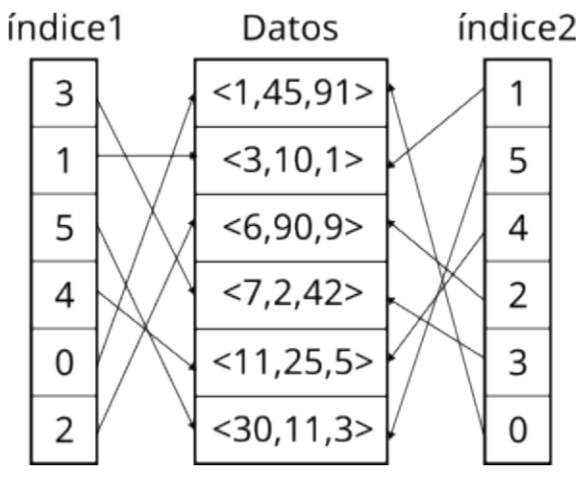
```
var nuevaListaEnlazada: ListaEnlazada<T>
    nuevaListaEnlazada:= new listaEnlazadaVacia()

var i: int
    i:= 0

while (i < c1.tamaño()) do
  if pertenece(c2, c1.datos.obtener(i)) then
    nuevaListaEnlazada.agregarAtras(c1.datos.obtener(i))
  endif
  i:= i + 1
endwhile

c.datos = nuevaListaEnlazada
c.tamaño = c.datos.tamaño()
```

}

Ejercicio 4

Ejercicio 4. Un *índice* es una estructura secundaria que permite acceder más rápidamente a los datos a partir de un determinado criterio. Básicamente un índice guarda *posiciones* o *punteros* a los elementos en un orden en particular, diferente al orden original.

Imagine una secuencia de tuplas con varias componentes, ordenada por su primer componente. Algunas veces vamos a querer buscar (rápido) por las demás componentes. Podríamos guardar los datos en sí en un arreglo y tener arreglos con las posiciones ordenadas por las demás componentes. A estos arreglos se los denomina índices.



En la figura, si recorremos los datos en el orden en el que están guardados, obtenemos:
`[<1, 45, 91>, <3, 10, 1>, <6, 90, 9>, <7, 2, 42>, <11, 25, 5>, <30, 11, 3>]`

Si lo recorremos usando el índice 1 (que apunta a los elementos en función de la segunda componente) obtenemos:
`[<7, 2, 42>, <3, 10, 1>, <30, 11, 3>, <11, 25, 5>, <1, 45, 91>, <6, 90, 9>]`

- Escriba la estructura propuesta
- Escriba el invariante de representación y la función de abstracción, en castellano y en lógica para el TAD $\mathsf{Conjunto}\langle\mathsf{Tupla}\langle\mathbb{Z},\mathbb{Z},\mathbb{Z}\rangle\rangle$
- Escriba el algoritmo de `BuscarPor` que busca por alguna componente
- Escriba los algoritmos de **`agregar`** y **`sacar`**

Figura 4: Enunciado Problema 4

`Tupla` $= Tuple\langle int, int, int\rangle$

```
Modulo ConjuntoRapido implements Conjunto<Tupla<ℤ, ℤ, ℤ>> {
    var elems: Array<Tupla>
    var ordenadosPorCoord0: Array<int>
    var ordenadosPorCoord1: Array<int>
    var ordenadosPorCoord2: Array<int>

    pred indicesValidos (c: ConjuntoRapido) {
        (∀j : ℤ) (j ∈ ordenadosPorCoord0 ↔ 0 ≤ j < length(c.elems)) ∧
        (∀j : ℤ) (j ∈ ordenadosPorCoord1 ↔ 0 ≤ j < length(c.elems)) ∧
        (∀j : ℤ) (j ∈ ordenadosPorCoord2 ↔ 0 ≤ j < length(c.elems))
    }
    pred sinRepetidosPorCoordenada (c: ConjuntoRapido) {
        (∀coord : ℤ) (0 ≤ coord ≤ 2 ⟶_L (
            (∀i, j : ℤ) ((0 ≤ i, j < length(c.elems) ∧_L c.elems[i][coord] = c.elems[j][coord]) ⟶ i = j)
        ))
    }
    pred indicesOrdenados (s: Array<int>, coord: int) {
        0 ≤ coord ≤ 2 ∧_L
        (∀j : ℤ) (0 ≤ j < (length(s) − 1) ⟶_L (
            c.elems[s[j]][coord] < c.elems[s[j + 1]][coord]
        ))
    }
    pred ordenados (c: ConjuntoRapido) {
        indicesOrdenados(ordenadosPorCoord0, 0) ∧
        indicesOrdenados(ordenadosPorCoord1, 1) ∧
        indicesOrdenados(ordenadosPorCoord2, 2)
    }

    % Los tipos viven en "mundos" distintos, pero no estoy seguro como hacerlo... %
    pred abs (c: ConjuntoRapido, c': Conjunto<Tupla<ℤ, ℤ ℤ>>) {
        |c'.elems| = length(c.elems) ∧
        (∀t : Tupla<ℤ, ℤ, ℤ>) (t ∈ c'.elems ↔ t ∈ c.elems)
    }
    pred invRep (c: ConjuntoRapido) {
        length(c.elems) = length(ordenadosPorCoord1) = length(ordenadosPorCoord2) ∧_L
        indicesValidos(c) ∧_L sinRepetidosPorCoordenada(c) ∧ ordenados(c)
    }

    proc obtenerOrdenadosPorCoord (in c: ConjuntoRapido, coord: int) : Array<int>
```

```
if coord == 0 then
  return c.ordenadosPorCoord0
endif

if coord == 1 then
  return c.ordenadosPorCoord1
endif

return c.ordenadosPorCoord2
```

```
proc actualizarOrdenadosPorCoord (
    inout c: ConjuntoRapido,
    coord: int,
    ordenadosPorCoord: Array<int>
)
```

```
if coord == 0 then
  c.ordenadosPorCoord0:= ordenadosPorCoord
  return
endif

if coord == 1 then
  c.ordenadosPorCoord1:= ordenadosPorCoord
  return
endif

c.ordenadosPorCoord2:= ordenadosPorCoord
return
```

```
proc buscarPorCoord (in c: ConjuntoRapido, in coord: int, in valorCoord: int) : Tupla
```

```
var largo: int
    ordenadosPorCoord: Array<int>
    largo:= length(ordenadosPorCoord)
    ordenadosPorCoord:= obtenerOrdenadosPorCoord(c, coord)

var low: int
var high: int
    low:= 0
    high:= largo - 1

while (low <= high && low < largo) do
    mid:= floor((low + high) / 2)
    pivot:= c.elems[ordenadosPorCoord[mid]][coord]

    if pivot < valorCoord then
        low:= mid + 1
    else
        high:= mid - 1
    endif
endwhile

return c.elems[ordenadosPorCoord[low]][coord]
```

proc agregarEnOrdenadosPorCoord (inout c: ConjuntoRapido, in coord: int, in valorCoord: int)

```
var ordenadosPorCoord: Array<int>
    ordenadosPorCoord:= obtenerOrdenadosPorCoord(c, coord)

// En este caso, elems tiene exactamente un elemento, y agrego su índice: 0.
if length(ordenadosPorCoord) == 0:
  actualizarOrdenadosPorCoord(c, coord, new Array<int>(1)[0])
  return
endif

var nuevoOrdenadosPorCoord: Array<int>
    nuevoOrdenadosPorCoord:= new Array<int>(length(ordenadosPorCoord) + 1)

// Si bien acá se podría hacer un búsqueda binaria para obtener
// el índice para la nueva posición, eso no reducirá la complejidad del algoritmo
var posicionValorCoord: int
    posicionValorCoord:= 0

while (c.elems[ordenadosPorCoord[posicionValorCoord]][coord] < valorCoord) do
  posicionValorCoord:= posicionValorCoord + 1
endwhile

// Copio la porción del arreglo que mantiene el orden inferior a posicionValorCoord
var i: int
    i:= 0

while (i < posicionValorCoord) do
  nuevosOrdenadosPorCoords[i]:= ordenadosPorCoord[i]
  i:= i + 1
endwhile

// Inserto el índice correspondiente al último elemento de c.elems
nuevosOrdenadosPorCoords[posicionValorCoord]:= length(c.elems) - 1

// Copio la porción del arreglo que mantiene el orden superior a posicionValorCoord
i:= posicionValorCoord + 1

while (j <= length(ordenadosPorCoord)) do
  nuevosOrdenadosPorCoords[i]:= ordenadosPorCoord[i - 1]
  i:= i + 1
endwhile

// Actualizo el arreglo correspondiente para esta coord
actualizarOrdenadosPorCoord(c, coord, nuevosOrdenadosPorCoords)

return
```

proc agregar (inout c: ConjuntoRapido, in t: Tupla)

```
var nuevosElems: Array<Tupla>
    nuevosElems:= new Array<Tupla>(length(c.elems) + 1)

var i: int
    i:= 0

while (i < length(c.elems)) do
  nuevosElems[i]:= c.elems[i]
  i:= i + 1
endwhile

nuevosElems[i]:= t

c.elems:= nuevosElems
agregarEnOrdenadosPorCoord(c, 0, t[0])
agregarEnOrdenadosPorCoord(c, 1, t[1])
agregarEnOrdenadosPorCoord(c, 2, t[2])

return
```

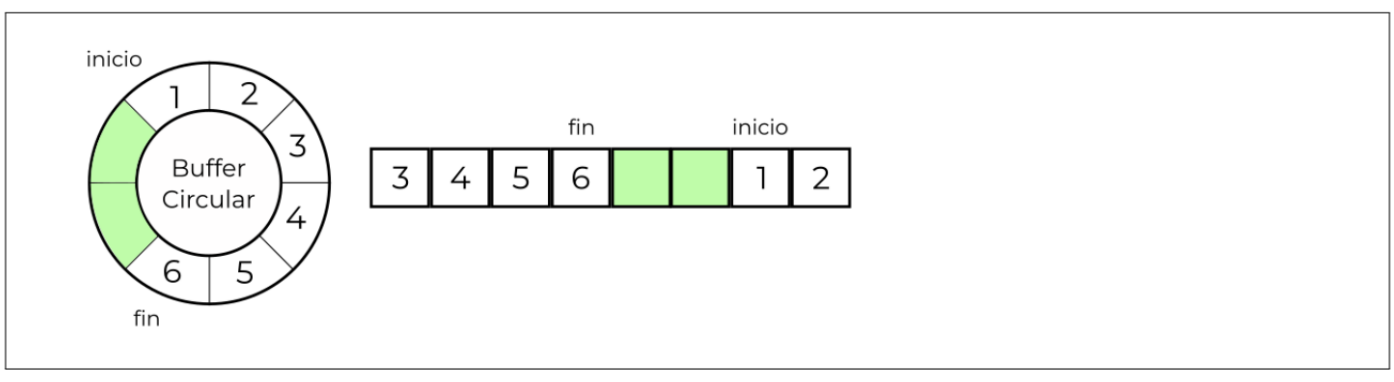
proc sacar (inout c: ConjuntoRapido, in t: Tupla) : T

```
// @todo
```

}

Ejercicio 5

Ejercicio 5. Una forma eficiente de implementar el TAD Cola en su versión acotada (con una cantidad máxima de elementos predefinida), es mediante un *buffer circular*. Esta estructura está formada por un array del tamaño máximo de la cola (n) y dos índices ($inicio$ y fin), para indicar adonde empieza y adonde termina la cola. El chiste de esta estructura es que, al llegar al final del arreglo, si los elementos del principio ya fueron consumidos, se puede reusar dichas posiciones.



- Elija una estructura de representación
- Escriba el invariante de representación y la función de abstracción
- Escriba los algoritmos de las operaciones **encolar** y **desencolar**
- ¿Por qué tiene sentido utilizar un buffer circular para una cola y no para una pila?

Figura 5: Enunciado Problema 5

```
Modulo ColaCirciular<T> implements ColaAcotada<T> {
    var elems: Array<T>
    var inicio: int
    var fin: int

    aux cantidadElementosDeInicioAFin (c: ColaCircular<T>) : ℤ = c.fin − c.inicio;
    aux cantidadElementosDeFinAInicio (c: ColaCircular<T>) : ℤ = (length(c.elems) − inicio) + (c.fin);
    aux cantidadElementos (c: ColaCircular<T>) : ℤ =
        if c.inicio < c.fin then cantidadElementosDeInicioAFin(p) else cantidadElementosDeFinAInicio(p) fi;

    pred abs (c: ColaCircular<T>, c': ColaAcotada<T>) {
        c'.cota = length(c.elems) ∧ |c'.s| = cantidadElementos(c) ∧
        (∀j : ℤ) (0 ≤ j < |c'.s| ⟶_L c'.s[j] = c.elems[(c.inicio + j) mod |c'.s|])
    }
    pred invRep (c: ColaCircular<T>) {
        (0 ≤ c.inicio, c.fin < length(c.elems)) ∧
        ((c.inicio < c.fin) ∨ (c.fin < c.inicio)) ∧_L
        (∀j : ℤ) (0 ≤ j < cantidadElementos(c) ⟶_L def(c.elems[j]))
    }

    proc encolar (inout c: ColaCircular<T>, in t: T)
```

```
1 c.elems[c.fin]:= t
2 c.fin:= (c.fin + 1) % length(c.elems)
3
4 return
```

```
    proc desencolar (inout c: ColaCircular<T>) : T
```

```
1 var elemento: T
2     elemento:= c.elems[c.inicio]
3
4 c.inicio:= (c.inicio + 1) % length(c.elems)
5
6 return elemento
```

```
}
```