# Apunte de Mï¿½dulos Bï¿½sicos (v. 0.3$\alpha$)

Algoritmos y Estructuras de Datos II, DC, UBA.

$1^{\text{er}}$ cuatrimestre de 2019

## Índice

# 1. Introducciï¿½n

El presente documento describe varios mï¿½dulos que se pueden utilizar para realizar el TP de diseï¿½o. Ademï¿½s, sirve como ejemplo de quï¿½ se espera del TP de diseï¿½o, y muestra algunas tï¿½cnicas que podrï¿½an ser ï¿½tiles a la hora de desarrollar nuevos mï¿½dulos.

Antes de introducir los mï¿½dulos, se especifican los tipos de iteradores que se van a utilizar. Esta especificaciï¿½n es auxiliar, para simplificar las precondiciones y postcondiciones de los algoritmos que utilizan iteradores. Luego, se presentan todos los mï¿½dulos, con su interfaz, representaciï¿½n y cï¿½lculos de complejidad.

**NOTA: Este apunte no estï¿½ terminado. Ademï¿½s de ser incompleto (faltan los algoritmos y los cï¿½lculos de complejidad de todos los mï¿½dulos), puede tener (mejor dicho, tiene) errores y podrï¿½a sufrir cambios en cualquier momento.**

# 2. TADs para especificar iteradores

En esta secciï¿½n se describen los TADs que utilizamos en la materia para especificar los iteradores. Los mismos no son mï¿½s que un conjunto de funciones auxiliares que sirven para especificar las precondiciones y postcondiciones de las funciones que involucran iteradores. La forma de especificar estos iteradores es "envolviendo" una estructura que representa el concepto de ordenamiento de los valores contenidos. En este sentido, la especificaciï¿½n de los iteradores con TADs podrï¿½a evitarse, pero lo incluimos para simplificar la especificaciï¿½n de los mï¿½dulos.

## 2.1. TAD Iterador Unidireccional($\alpha$)

El iterador unidireccional permite recorrer los elementos una ï¿½nica vez, avanzando continuamente. Es el tipo de iterador mï¿½s simple que se puede especificar y no permite modificar la estructura iterada. Como la idea es convertir cualquier estructura en una secuencia, es razonable que este iterador tome una secuencia en la parte de especificaciï¿½n. La idea final es que esta secuencia describa el orden en el que se recorrerï¿½n los elementos de la estructura, i.e., esta secuencia es una "permutaciï¿½n" de la estructura iterada.

**TAD** Iterador Unidireccional($\alpha$)

**parámetros formales**
**gï¿½neros** $\alpha$

**géneros** itUni($\alpha$)

**igualdad observacional**
$(\forall it_1, it_2 : \text{it}(\alpha))\ \ (it_1 =_{\text{obs}} it_2 \iff (\text{Siguientes}(it_1) =_{\text{obs}} \text{Siguientes}(it_2)))$

**observadores básicos**
Siguientes : itUni($\alpha$) $\longrightarrow$ secu($\alpha$)

**generadores**
CrearItUni : secu($\alpha$) $\longrightarrow$ itUni($\alpha$)

**otras operaciones**
HayMas? : itUni($\alpha$) $\longrightarrow$ bool
Actual : itUni($\alpha$) $it$ $\longrightarrow$ $\alpha$ {HayMas?($it$)}
Avanzar : itUni($\alpha$) $it$ $\longrightarrow$ itUni($\alpha$) {HayMas?($it$)}

**axiomas**
Siguientes(CrearItUni($i$)) $\equiv$ $i$
HayMas?($it$) $\equiv$ $\neg$Vacia?(Siguientes($it$))
Actual($it$) $\equiv$ Prim(Siguientes($it$))
Avanzar($it$) $\equiv$ CrearItUni(Fin(Siguientes($it$)))

**Fin TAD**

## 2.2. TAD Iterador Unidireccional Modificable($\alpha$)

El iterador unidireccional modificable es una extensiï¿½n del iterador unidireccional que permite realizar algunas operaciones de modificaciï¿½n sobre los elementos de la estructura recorrida. Para poder especificar las modificaciones a la estructura iterada, se guarda la secuencia de los elementos que ya fueron recorridos. Observar que para especificar los efectos secundarios que tienen estas modificaciones en el tipo iterado, hay que aclarar cï¿½mo es el aliasing entre el iterador y el tipo iterado en el mï¿½dulo correspondiente.

**TAD** Iterador Unidireccional Modificable($\alpha$)

**parámetros formales**

**gï¿½neros** $\alpha$

**géneros** itMod($\alpha$)

**igualdad observacional**

$$(\forall it_1, it_2 : \text{itMod}(\alpha)) \left( it_1 =_{\text{obs}} it_2 \iff \begin{pmatrix} \text{Anteriores}(it_1) =_{\text{obs}} \text{Anteriores}(it_2) \land \\ \text{Siguientes}(it_1) =_{\text{obs}} \text{Siguientes}(it_2) \end{pmatrix} \right)$$

**observadores básicos**

Anteriores : itMod($\alpha$) $\longrightarrow$ secu($\alpha$)
Siguientes : itMod($\alpha$) $\longrightarrow$ secu($\alpha$)

**generadores**

CrearItMod : secu($\alpha$) $\times$ secu($\alpha$) $\longrightarrow$ itMod($\alpha$)

**otras operaciones**

| | | | |
|---|---|---|---|
| SecuSuby | : itMod($\alpha$) | $\longrightarrow$ secu($\alpha$) | |
| HayMas? | : itMod($\alpha$) | $\longrightarrow$ bool | |
| Actual | : itMod($\alpha$) *it* | $\longrightarrow$ $\alpha$ | {HayMas?(*it*)} |
| Avanzar | : itMod($\alpha$) *it* | $\longrightarrow$ itMod($\alpha$) | {HayMas?(*it*)} |
| Eliminar | : itMod($\alpha$) *it* | $\longrightarrow$ itMod($\alpha$) | {HayMas?(*it*)} |
| Agregar | : itMod($\alpha$) $\times$ $\alpha$ | $\longrightarrow$ itMod($\alpha$) | |

**axiomas**

| | | |
|---|---|---|
| Anteriores(CrearItMod($i$, $d$)) | $\equiv$ | $i$ |
| Siguientes(CrearItMod($i$, $d$)) | $\equiv$ | $d$ |
| SecuSuby($it$) | $\equiv$ | Anteriores($it$) & Siguientes($it$) |
| HayMas?($it$) | $\equiv$ | $\neg$Vacia?(Siguientes($it$)) |
| Actual($it$) | $\equiv$ | Prim(Siguientes($it$)) |
| Avanzar($it$) | $\equiv$ | CrearItMod(Anteriores($it$) $\circ$ Actual($it$), Fin(Siguientes($it$))) |
| Eliminar($it$) | $\equiv$ | CrearItMod(Anteriores($it$), Fin(Siguientes($it$))) |
| Agregar($it, a$) | $\equiv$ | CrearItMod(Anteriores($it$) $\circ$ $a$, Siguientes($it$)) |

**Fin TAD**

## 2.3. Iterador Bidireccional($\alpha$)

El iterador bidireccional es una generalizaciï¿½n del iterador unidireccional modificable. El mismo permite recorrer los elementos avanzando y retrocediendo. Si bien se podrï¿½a hacer una versiï¿½n de iterador bidireccional no modificable, la especificaciï¿½n de ambas es similar. Cuando se utilice en un mï¿½dulo que no permita algunas modificaciones, simplemente se puede omitir el diseï¿½o de las funciones que realizan estas modificaciones (ver e.g., mï¿½dulo Conjunto Lineal). Por este motivo, optamos sï¿½lo por la versiï¿½n modificable.

Para que el iterador bidireccional sea lo mas simï¿½trico posible, cambiamos la operaciï¿½n actual por dos: anterior y siguiente. La idea conceptual es pensar que el iterador estï¿½ posicionado en el medio de dos posiciones, y puede acceder tanto a la anterior como a la siguiente. Obviamente, la implementaciï¿½n puede diferir de esta visiï¿½n conceptual.

**TAD** Iterador Bidireccional($\alpha$)

**parámetros formales**

**gï¿½neros** $\alpha$

**géneros** itBi($\alpha$)

**igualdad observacional**

$$(\forall it_1, it_2 : \text{itBi}(\alpha)) \left( it_1 =_{\text{obs}} it_2 \iff \begin{pmatrix} \text{Anteriores}(it_1) =_{\text{obs}} \text{Anteriores}(it_2) \land \\ \text{Siguientes}(it_1) =_{\text{obs}} \text{Siguientes}(it_2) \end{pmatrix} \right)$$

**observadores básicos**

Anteriores : itBi($\alpha$) $\longrightarrow$ secu($\alpha$)

Siguientes : itBi($\alpha$) $\longrightarrow$ secu($\alpha$)

**generadores**

CrearItBi : secu($\alpha$) $\times$ secu($\alpha$) $\longrightarrow$ itBi($\alpha$)

**otras operaciones**

| | | | | |
|---|---|---|---|---|
| SecuSuby | : itBi($\alpha$) | $\longrightarrow$ | secu($\alpha$) | |
| HayAnterior? | : itBi($\alpha$) | $\longrightarrow$ | bool | |
| Anterior | : itBi($\alpha$) $it$ | $\longrightarrow$ | $\alpha$ | {HayAnterior?($it$)} |
| Retroceder | : itBi($\alpha$) $it$ | $\longrightarrow$ | itBi($\alpha$) | {HayAnterior?($it$)} |
| HaySiguiente? | : itBi($\alpha$) | $\longrightarrow$ | bool | |
| Siguiente | : itBi($\alpha$) $it$ | $\longrightarrow$ | $\alpha$ | {HaySiguiente?($it$)} |
| Avanzar | : itBi($\alpha$) $it$ | $\longrightarrow$ | itBi($\alpha$) | {HaySiguiente?($it$)} |
| EliminarSiguiente | : itBi($\alpha$) $it$ | $\longrightarrow$ | itBi($\alpha$) | {HaySiguiente?($it$)} |
| EliminarAnterior | : itBi($\alpha$) $it$ | $\longrightarrow$ | itBi($\alpha$) | {HayAnterior?($it$)} |
| AgregarComoAnterior | : itBi($\alpha$) $\times$ $\alpha$ | $\longrightarrow$ | itBi($\alpha$) | |
| AgregarComoSiguiente | : itBi($\alpha$) $\times$ $\alpha$ | $\longrightarrow$ | itBi($\alpha$) | |

**axiomas**

| | | |
|---|---|---|
| Anteriores(CrearItBi($i$, $d$)) | $\equiv$ | $i$ |
| Siguientes(CrearItBi($i$, $d$)) | $\equiv$ | $d$ |
| SecuSuby($it$) | $\equiv$ | Anteriores($i$) & Siguientes($d$) |
| HayAnterior?($it$) | $\equiv$ | $\neg$Vacia?(Anteriores($it$)) |
| Anterior($it$) | $\equiv$ | Ult(Anteriores($it$)) |
| Retroceder($it$) | $\equiv$ | CrearItBi(Com(Anteriores($it$)), Anterior($it$) $\bullet$ Siguientes($it$)) |
| HaySiguiente?($it$) | $\equiv$ | $\neg$Vacia?(Siguientes($it$)) |
| Siguiente($it$) | $\equiv$ | Prim(Siguientes($it$)) |
| Avanzar($it$) | $\equiv$ | CrearItBi(Anteriores($it$) $\circ$ Siguiente($it$), Fin(Siguientes($it$))) |
| EliminarSiguiente($it$) | $\equiv$ | CrearItBi(Anteriores($it$), Fin(Siguientes($it$))) |
| EliminarAnterior($it$) | $\equiv$ | CrearItBi(Com(Anteriores($it$)), Siguientes($it$)) |
| AgregarComoAnterior($it, a$) | $\equiv$ | CrearItBi(Anteriores($it$) $\circ$ $a$, Siguientes($it$)) |
| AgregarComoSiguiente($it, a$) | $\equiv$ | CrearItBi(Anteriores($it$), $a$ $\bullet$ Siguientes($it$)) |
| SecuSuby($it$) | $\equiv$ | Anteriores($it$) & Siguientes($it$) |

**Fin TAD**

# 3. Invariantes de aliasing

Para simplificar la descripciï¿½n del aliasing entre dos variables, vamos a definir un "metapredicado". Este metapredicado, llamado *alias*, lo vamos a utilizar para describir aquellas variables que comparten memoria en la ejecuciï¿½n del programa. Si bien el metapredicado alias no es parte del lenguaje de TADs y no lo describimos en lï¿½gica de primer orden, lo vamos a utilizar en las precondiciones y postcondiciones de las funciones. En esta secciï¿½n vamos a describir su semï¿½ntica en castellano.

Alias es un metapredicado con un ï¿½nico parï¿½metro $\phi$ que puede ser una expresiï¿½n booleana del lenguaje de TADs o un predicado en lï¿½gica de primer orden. Este paramï¿½tro $\phi$ involucrarï¿½ un conjunto $V$ con dos o mï¿½s variables del programa. El significado es que las variables de $V$ satisfacen $\phi$ durante la ejecuciï¿½n del resto del programa, siempre y cuando dichas variables no sean asignadas con otro valor. En particular, el invariante puede dejar de satisfacerse cuando una variable de $V$ se indefine. Una variable se indefine, cuando el valor al que hace referencia deja de ser valido. Esto ocurre principalmente cuando se elimina un elemento que estï¿½ siendo iterado.

Por ejemplo, supongamos que $s$ y $t$ son dos variables de tipo $\alpha$. Si escribimos

$$\text{alias}(s = t),$$

lo que significa informalmente es que $s$ y $t$ comparten la misma posiciï¿½n de memoria. Un poco mï¿½s rigurosamente, lo que significa es que cualquier modificaciï¿½n que se realice a $s$ afecta a $t$ y viceversa, de forma tal que $s = t$, mientras a $s$ y a $t$ no se les asigne otro valor.

El ejemplo anterior es un poco bï¿½sico. Supongamos ahora que tenemos dos variables $s$ y $c$ de tipos secu($\alpha$) y conj($\alpha$), respectivamente. Si escribimos

$$\text{alias}(\text{esPermutacion}(s, c)),$$

estamos diciendo que $s$ y $c$ comparten la misma memoria de forma tal que cualquier modificacii¿½n sobre $s$ afecta a $c$ y viceversa, de forma tal que se satisface esPermutacion$(s, c)$. En particular, si se agrega un elemento $a$ a $c$, se obtiene que la secuencia $s$ se modifica de forma tal que resulta una permutacii¿½n de $c \cup \{a\}$. Notemos que, en particular, $s$ podrï¿½a cambiar a cualquier permutacii¿½n, salvo que se indique lo contrario. De la misma forma, si se eliminara un elemento $a$ de $s$, entonces $c$ tambien se verï¿½a afectado de forma tal que $s$ sea una permutacii¿½n de $c$. En particular, $c$ pasarï¿½a a ser $c \setminus \{a\}$.

Debemos observar que este invariante no es magico, sino que es una declaracii¿½n como cualquier otra, y el programado debe asegurarse que este invariante se cumpla. En particular, en el ejemplo anterior, no deberiamos permitir la inserciï¿½n de elementos repetido en $s$, ya que dejarï¿½a de ser una posible permutacii¿½n de un conjunto.

# 4. Modulo Mapa

# Interfaz

**se explica con**: Habitacion.

**generos**: `mapa`.

## Operaciones basicas de mapa

nuevoMapa(**in** $n$: `nat`) $\rightarrow$ $res$ : `mapa`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} nuevaHab(n)\}$
**Complejidad:** TODO
**Descripcii¿½n:** genera un mapa de tamano n x n.

ocupar(**in** $m$: `mapa`, **in** $c$: $tupla(int, int)$) $\rightarrow$ $res$ : `mapa`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{c \in casilleros(m) \wedge_L libre(m, c) \wedge \}$

AgregarAdelante(**in/out** $l$: `lista(`$\alpha$`)`, **in** $a$: $\alpha$) $\rightarrow$ $res$ : `itLista(`$\alpha$
**Pre** $\equiv$ $\{l =_{\text{obs}} l_0\}$
**Post** $\equiv$ $\{l =_{\text{obs}} a \bullet l_0 \wedge res = \text{CrearItBi}(<>, l) \wedge \text{alias}(\text{SecuSuby}(res) = l)\}$
**Complejidad:** $\Theta(copy(a))$
**Descripcii¿½n:** agrega el elemento $a$ como primer elemento de la lista. Retorna un iterador a $l$, de forma tal que Siguiente devuelva $a$.
**Aliasing:** el elemento $a$ agrega por copia. El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funcii¿½n EliminarSiguiente.

AgregarAtras(**in/out** $l$: `lista(`$\alpha$`)`, **in** $a$: $\alpha$) $\rightarrow$ $res$ : `itLista(`$\alpha$`)`
**Pre** $\equiv$ $\{l =_{\text{obs}} l_0\}$
**Post** $\equiv$ $\{l =_{\text{obs}} l_0 \circ a \wedge res = \text{CrearItBi}(l_0, a) \wedge \text{alias}(\text{SecuSuby}(res) = l)\}$
**Complejidad:** $\Theta(copy(a))$
**Descripcii¿½n:** agrega el elemento $a$ como ï¿½ltimo elemento de la lista. Retorna un iterador a $l$, de forma tal que Siguiente devuelva $a$.
**Aliasing:** el elemento $a$ se agrega por copia. El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funcii¿½n EliminarSiguiente.

EsVacï¿½a?(**in** $l$: `lista(`$\alpha$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \text{vacia?}(l)\}$
**Complejidad:** $\Theta(1)$
**Descripcii¿½n:** devuelve `true` si y sï¿½lo si $l$ no contiene elementos

Fin(**in/out** $l$: `lista(`$\alpha$`)`)
**Pre** $\equiv$ $\{l =_{\text{obs}} l_0 \wedge \neg\text{vacï¿½a?}(l)\}$
**Post** $\equiv$ $\{l =_{\text{obs}} \text{fin}(l_0)\}$
**Complejidad:** $\Theta(1)$
**Descripcii¿½n:** elimina el primer elemento de $l$

Comienzo(**in/out** $l$ : lista($\alpha$))
**Pre** ≡ {$l =_{\text{obs}} l_0 \wedge \neg$vacï¿½a?($l$)}
**Post** ≡ {$l =_{\text{obs}}$ com($l_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** elimina el ï¿½ltimo elemento de $l$

Primero(**in** $l$ : lista($\alpha$)) → *res* : $\alpha$
**Pre** ≡ {$\neg$vacï¿½a?($l$)}
**Post** ≡ {alias($res =_{\text{obs}}$ prim($l$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el primer elemento de la lista.
**Aliasing:** *res* es modificable si y sï¿½lo si $l$ es modificable.

Ultimo(**in** $l$ : lista($\alpha$)) → *res* : $\alpha$
**Pre** ≡ {$\neg$vacï¿½a?($l$)}
**Post** ≡ {alias($res =_{\text{obs}}$ ult($l$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el ï¿½ltimo elemento de la lista.
**Aliasing:** *res* es modificable si y sï¿½lo si $l$ es modificable.

Longitud(**in** $l$ : lista($\alpha$)) → *res* : nat
**Pre** ≡ {true}
**Post** ≡ {$res =_{\text{obs}}$ long($l$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve la cantidad de elementos que tiene la lista.

**•[•]**(**in** $l$ : lista($\alpha$), **in** $i$ : nat) → *res* : $\alpha$
**Pre** ≡ {$i <$ long($l$)}
**Post** ≡ {alias($res =_{\text{obs}}$ iesimo($l$, $i$))}
**Complejidad:** $\Theta(i)$
**Descripciï¿½n:** devuelve el elemento que se encuentra en la $i$-ï¿½sima posiciï¿½n de la lista en base 0. Es decir, $l[i]$ devuelve el elemento que se encuentra en la posiciï¿½n $i+1$.
**Aliasing:** *res* es modificable si y sï¿½lo si $l$ es modificable.

Copiar(**in** $l$ : lista($\alpha$)) → *res* : lista($\alpha$)
**Pre** ≡ {true}
**Post** ≡ {$res =_{\text{obs}} l$}
**Complejidad:** $\Theta\left(\sum_{i=1}^{\ell} copy(l[i])\right)$, donde $\ell =$ long($l$).
**Descripciï¿½n:** genera una copia nueva de la lista.

**• = •**(**in** $l_1$ : lista($\alpha$), **in** $l_2$ : lista($\alpha$)) → *res* : bool
**Pre** ≡ {true}
**Post** ≡ {$res =_{\text{obs}} l_1 = l_2$}
**Complejidad:** $\Theta\left(\sum_{i=1}^{\ell} equal(l_1[i], l_2[i])\right)$, donde $\ell = \text{mín}\{\text{long}(l_1), \text{long}(l_2)\}$.
**Descripciï¿½n:** compara $l_1$ y $l_2$ por igualdad, cuando $\alpha$ posee operaciï¿½n de igualdad.
**Requiere:** **• = •**(**in** $a_1$ : $\alpha$, **in** $a_2$ : $\alpha$) → *res* : bool
**Pre** ≡ {true}
**Post** ≡ {$res =_{\text{obs}} (a_1 = a_2)$}
**Complejidad:** $\Theta(equal(a_1, a_2))$
**Descripciï¿½n:** funciï¿½n de igualdad de $\alpha$'s

## Operaciones del iterador

El iterador que presentamos permite modificar la lista recorrida. Sin embargo, cuando la lista es no modificable, no se pueden utilizar las funciones que la modificarï¿½an, teniendo en cuenta el aliasing existente entre el iterador y la lista iterada. Cuando la lista es modificable, vamos a decir que el iterador generado es modificable.

CrearIt(**in** $l$ : `lista(`$\alpha$`)`) $\rightarrow$ $res$ : `itLista(`$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ crearItBi(<>, $l$) $\wedge$ alias(SecuSuby($it$) = $l$)$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** crea un iterador bidireccional de la lista, de forma tal que al pedir Siguiente se obtenga el primer elemento de $l$.
**Aliasing:** el iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n EliminarSiguiente.

CrearItUlt(**in** $l$ : `lista(`$\alpha$`)`) $\rightarrow$ $res$ : `itLista(`$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ crearItBi($l$, <>) $\wedge$ alias(SecuSuby($it$) = $l$)$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** crea un iterador bidireccional de la lista, de forma tal que al pedir Anterior se obtenga el ï¿½ltimo elemento de $l$.
**Aliasing:** el iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n EliminarSiguiente.

HaySiguiente(**in** $it$ : `itLista(`$\alpha$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ haySiguiente?($it$)$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si en el iterador todavï¿½a quedan elementos para avanzar.

HayAnterior(**in** $it$ : `itLista(`$\alpha$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ hayAnterior?($it$)$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si en el iterador todavï¿½a quedan elementos para retroceder.

Siguiente(**in** $it$ : `itLista(`$\alpha$`)`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {HaySiguiente?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Siguiente($it$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento siguiente a la posiciï¿½n del iterador.
**Aliasing:** $res$ es modificable si y sï¿½lo si $it$ es modificable.

Anterior(**in** $it$ : `itLista(`$\alpha$`)`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {HayAnterior?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Anterior($it$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento anterior del iterador.
**Aliasing:** $res$ es modificable si y sï¿½lo si $it$ es modificable.

Avanzar(**in/out** $it$ : `itLista(`$\alpha$`)`)
**Pre** $\equiv$ $\{it = it_0 \wedge$ HaySiguiente?($it$)$\}$
**Post** $\equiv$ $\{it =_{\text{obs}}$ Avanzar($it_0$)$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** avanza el iterador a la posiciï¿½n siguiente.

Retroceder(**in/out** $it$ : `itLista(`$\alpha$`)`)
**Pre** $\equiv$ $\{it = it_0 \wedge$ HayAnterior?($it$)$\}$
**Post** $\equiv$ $\{it =_{\text{obs}}$ Retroceder($it_0$)$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** retrocede el iterador a la posiciï¿½n anterior.

EliminarSiguiente(**in/out** $it$ : `itLista(`$\alpha$`)`)
**Pre** $\equiv$ $\{it = it_0 \wedge$ HaySiguiente?($it$)$\}$
**Post** $\equiv$ $\{it =_{\text{obs}}$ EliminarSiguiente($it_0$)$\}$
**Complejidad:** $\Theta(1)$

**Descripciï¿½n:** elimina de la lista iterada el valor que se encuentra en la posiciï¿½n siguiente del iterador.

EliminarAnterior(**in/out** *it*: `itLista(`$\alpha$`)`)
**Pre** $\equiv$ $\{it = it_0 \wedge \text{HayAnterior?}(it)\}$
**Post** $\equiv$ $\{it =_{\text{obs}} \text{EliminarAnterior}(it_0)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** elimina de la lista iterada el valor que se encuentra en la posiciï¿½n anterior del iterador.

AgregarComoSiguiente(**in/out** *it*: `itLista(`$\alpha$`)`, **in** *a*: $\alpha$)
**Pre** $\equiv$ $\{it = it_0\}$
**Post** $\equiv$ $\{it =_{\text{obs}} \text{AgregarComoSiguiente}(it_0, a)\}$
**Complejidad:** $\Theta(copy(a))$
**Descripciï¿½n:** agrega el elemento $a$ a la lista iterada, entre las posiciones anterior y siguiente del iterador, dejando al iterador posicionado de forma tal que al llamar a Siguiente se obtenga $a$.
**Aliasing:** el elemento $a$ se agrega por copia.

AgregarComoAnterior(**in/out** *it*: `itLista(`$\alpha$`)`, **in** *a*: $\alpha$)
**Pre** $\equiv$ $\{it = it_0\}$
**Post** $\equiv$ $\{it =_{\text{obs}} \text{AgregarComoAnterior}(it_0, a)\}$
**Complejidad:** $\Theta(copy(a))$
**Descripciï¿½n:** agrega el elemento $a$ a la lista iterada, entre las posiciones anterior y siguiente del iterador, dejando al iterador posicionado de forma tal que al llamar a Anterior se obtenga $a$.
**Aliasing:** el elemento $a$ se agrega por copia.

# Representaciï¿½n

## Representaciï¿½n de la lista

El objetivo de este mï¿½dulo es implementar una lista doblemente enlazada con punteros al principio y al fin. Para simplificar un poco el manejo de la estructura, vamos a reemplazarla por una lista circular, donde el siguiente del ï¿½ltimo apunta al primero y el anterior del primero apunta al ï¿½ltimo. La estructura de representaciï¿½n, su invariante de representaciï¿½n y su funciï¿½n de abstracciï¿½n son las siguientes.

**`lista`($\alpha$) se representa con `lst`**

donde `lst` es `tupla`(*primero*: `puntero(nodo)`, *longitud*: `nat`)
donde `nodo` es `tupla`(*dato*: $\alpha$, *anterior*: `puntero(nodo)`, *siguiente*: `puntero(nodo)`)

$\text{Rep} : \text{lst} \longrightarrow \text{bool}$
$\text{Rep}(l) \equiv \text{true} \iff (l.\text{primero} = \text{NULL}) = (l.\text{longitud} = 0) \wedge_{\text{L}} (l.\text{longitud} \neq 0 \Rightarrow_{\text{L}}$
$\text{Nodo}(l, l.\text{longitud}) = l.\text{primero} \wedge$
$(\forall i\colon \text{nat})(\text{Nodo}(l,i)\to\text{siguiente} = \text{Nodo}(l,i+1)\to\text{anterior}) \wedge$
$(\forall i\colon \text{nat})(1 \leq i < l.\text{longitud} \Rightarrow \text{Nodo}(l,i) \neq l.\text{primero})$

$\text{Nodo} : \text{lst } l \times \text{nat} \longrightarrow \text{puntero(nodo)}$ $\{l.\text{primero} \neq \text{NULL}\}$
$\text{Nodo}(l,i) \equiv$ **if** $i = 0$ **then** $l.\text{primero}$ **else** $\text{Nodo}(\text{FinLst}(l), i-1)$ **fi**

$\text{FinLst} : \text{lst} \longrightarrow \text{lst}$
$\text{FinLst}(l) \equiv \text{Lst}(l.\text{primero}\to\text{siguiente}, l.\text{longitud} - \text{mín}\{l.\text{longitud}, 1\})$

$\text{Lst} : \text{puntero(nodo)} \times \text{nat} \longrightarrow \text{lst}$
$\text{Lst}(p,n) \equiv \langle p, n\rangle$

$\text{Abs} : \text{lst } l \longrightarrow \text{secu}(\alpha)$ $\{\text{Rep}(l)\}$
$\text{Abs}(l) \equiv$ **if** $l.\text{longitud} = 0$ **then** $<>$ **else** $l.\text{primero}\to\text{dato} \bullet \text{Abs}(\text{FinLst}(l))$ **fi**

## Representaciï¿½n del iterador

El iterador es simplemente un puntero al nodo siguiente. Este puntero apunta a NULL en el caso en que se llegï¿½ al final de la lista. Por otra parte, el nodo anterior se obtiene accediendo al nodo siguiente y retrocediendo (salvo que el nodo siguiente sea el primer nodo). Para poder modificar la lista, tambien hay que guardar una referencia a la lista que estï¿½ siendo iterada. Ademï¿½s, de esta forma podemos saber si el iterador apunta al primero o no.

**`itLista`($\alpha$) se representa con `iter`**

donde `iter` es `tupla`(*siguiente*: `puntero(nodo)`, *lista*: `puntero(lst)`)

Rep : iter $\longrightarrow$ bool
Rep($it$) $\equiv$ true $\Longleftrightarrow$ Rep($*$($it$.lista)) $\wedge_{\text{L}}$ ($it$.siguiente = NULL $\vee_{\text{L}}$ ($\exists i$: nat)(Nodo($*it$.lista, $i$) = $it$.siguiente)

Abs : iter $it$ $\longrightarrow$ itBi($\alpha$) {Rep($it$)}
Abs($it$) $=_{\text{obs}}$ b: itBi($\alpha$) | Siguientes($b$) = Abs(Sig($it$.lista, $it$.siguiente)) $\wedge$
Anteriores($b$) = Abs(Ant($it$.lista, $it$.siguiente))

Sig : puntero(lst) $l$ $\times$ puntero(nodo) $p$ $\longrightarrow$ lst {Rep($\langle l, p\rangle$)}
Sig($i,p$) $\equiv$ Lst($p$, $l\rightarrow$longitud $-$ Pos($*l$, $p$))

Ant : puntero(lst) $l$ $\times$ puntero(nodo) $p$ $\longrightarrow$ lst {Rep($\langle l, p\rangle$)}
Ant($i,p$) $\equiv$ Lst(**if** $p = l\rightarrow$primero **then** NULL **else** $l\rightarrow$primero **fi**, Pos($*l$, $p$))

Nota: cuando $p$ = NULL, Pos devuelve la longitud de la lista, lo cual estï¿½ bien, porque significa que el iterador no tiene siguiente.

Pos : lst $l$ $\times$ puntero(nodo) $p$ $\longrightarrow$ puntero(nodo) {Rep($\langle l, p\rangle$)}
Pos($l$,$p$) $\equiv$ **if** $l$.primero = $p$ $\vee$ $l$.longitud = 0 **then** 0 **else** 1 + Pos(FinLst($l$), $p$) **fi**

# Algoritmos

En esta secciï¿½n se hace abuso de notaciï¿½n en los cï¿½lculos de ï¿½lgebra de ï¿½rdenes presentes en la justificaciones de los algoritmos. La operaciï¿½n de suma "+" denota secuencializaciï¿½n de operaciones con determinado orden de complejidad, y el sï¿½mbolo de igualdad "=" denota la pertenencia al orden de complejidad resultante.

## Algoritmos del mï¿½dulo

---

**iVacï¿½a**() $\rightarrow$ *res* : lst

$res \leftarrow \langle NULL, 0\rangle$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

---

**iAgregarAdelante**(**in/out** $l$: `lst`, **in** $a$: $\alpha$) $\rightarrow$ *res* : *iter*

$it \leftarrow CrearIt(l)$ $\triangleright\ \Theta(1)$
$AgregarComoSiguiente(it, a)$ $\triangleright\ \Theta(copy(a))$
$res \leftarrow it$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(copy(a))$
Justificaciï¿½n: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(copy(a))$. Aplicando ï¿½lgebra de ï¿½rdenes:
$\Theta(1) + \Theta(1) + \Theta(copy(a)) = \Theta(copy(a))$

---

**iAgregarAtras**(**in/out** $l$: `lst`, **in** $a$: $\alpha$) $\rightarrow$ *res* : *iter*

$it \leftarrow CrearItUlt(l)$ $\triangleright\ \Theta(1)$
$AgregarComoSiguiente(it, a)$ $\triangleright\ \Theta(copy(a))$
$res \leftarrow it$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(copy(a))$
Justificaciï¿½n: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(copy(a))$. Aplicando ï¿½lgebra de ï¿½rdenes: $\Theta(1)+\Theta(copy(a))+\Theta(1) = \Theta(copy(a))$

---

**iEsVacï¿½a?**(**in** $l$ : `lst`) $\rightarrow$ $res$ : $bool$

```
res ← (l.primero = NULL)                                   ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

**iFin**(**in/out** $l$ : `lst`)

```
CrearIt(l).EliminarSiguiente()                             ▷ Θ(1) + Θ(1)
```

Complejidad: $\Theta(1)$
Justificacïż½n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

**iComienzo**(**in/out** $l$ : `lst`)

```
CrearItUlt(l).EliminarAnterior()                           ▷ Θ(1) + Θ(1)
```

Complejidad: $\Theta(1)$
Justificacïż½n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

**iPrimero**(**in** $l$ : `lst`) $\rightarrow$ $res$ : $\alpha$

```
res ← CrearIt(l).Siguiente()                               ▷ Θ(1) + Θ(1)
```

Complejidad: $\Theta(1)$
Justificacïż½n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

**iï¿½ltimo**(**in** $l$ : `lst`) $\rightarrow$ $res$ : $\alpha$

```
res ← CrearItUlt(l).Anterior()                             ▷ Θ(1) + Θ(1)
```

Complejidad: $\Theta(1)$
Justificacïż½n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

**iLongitud**(**in** $l$ : `lst`) $\rightarrow$ $res$ : $nat$

```
res ← l.longitud                                           ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

**•[•]**(**in** $l$ : `lst`, **in** $i$ : $nat$) $\rightarrow$ $res$ : $\alpha$

```
it ← CrearIt(l)                                            ▷ Θ(1)
indice ← 0                                                 ▷ Θ(1)
while indice < i do                                        ▷ Θ(i)
    Avanzar(it)                                            ▷ Θ(1)
    indice ← indice + 1                                    ▷ Θ(1)
end while
res ← Siguiente(it)                                        ▷ Θ(1)
```

Complejidad: $\Theta(i)$
Justificacïż½n: El algoritmo tiene un ciclo que se va a repetir $i$ veces. En cada ciclo se hacen realizan funciones con costo $\Theta(1)$. Aplicando ïż½lgebra de ïż½rdenes sabemos que el ciclo tiene un costo total del orden $\Theta(i)$. El costo total del algoritmo serïż½ de: $\Theta(1) + \Theta(1) + \Theta(i)$*($\Theta(1)$+$\Theta(1)$) + $\Theta(1) = \Theta(i)$

**iCopiar**(**in** $l$ : `lst`) $\rightarrow res : lst$

$res \leftarrow Vacia()$ $\triangleright\ \Theta(1)$
$it \leftarrow CrearIt(l)$ $\triangleright\ \Theta(1)$
**while** $HaySiguiente(it)$ **do** $\triangleright\ \Theta(long(l))$
  $AgregarAtras(res, Siguiente(it))$ $\triangleright\ \Theta(copy(Siguiente(it)))$
  $Avanzar(it)$ $\triangleright\ \Theta(1)$
**end while**

Complejidad: $\Theta\left(\sum_{i=1}^{long(l)} copy(l[i])\right)$

Justificaciï¿½n: El algoritmo cuenta con un ciclo que se repetirï¿½ long(l) veces (recorre la lista entera). Por cada ciclo realiza una copia del elemento, el costo serï¿½ el de copiar el elemento. Por lo tanto, el costo total del ciclo serï¿½ la suma de copiar cada uno de los elementos de la lista. El resto de las llamadas a funciones tiene costo $\Theta(1)$. Por lo tanto el costo total es de: $\Theta(1) + \Theta(1) + \Theta(long(l)) * (\Theta(copy(Siguiente(it))) + \Theta(1)\ ) = \Theta\left(\sum_{i=1}^{long(l)} copy(l[i])\right)$

---

$\bullet =_i \bullet$(**in** $l_1$ : `lst`, **in** $l_2$ : `lst`) $\rightarrow res : bool$

$it_1 \leftarrow CrearIt(l_1)$ $\triangleright\ \Theta(1)$
$it_2 \leftarrow CrearIt(l_2)$ $\triangleright\ \Theta(1)$
**while** $HaySiguiente(it_1) \wedge HaySiguiente(it_2) \wedge Siguiente(it_1) = Siguiente(it_2)$ **do** $\triangleright$ [*]
  $Avanzar(it_1)$ // $\Theta(1)$
  $Avanzar(it_2)$ // $\Theta(1)$
**end while**
$res \leftarrow \neg(HaySiguiente(it_1) \vee HaySiguiente(it_2))$ $\triangleright\ \Theta(1) + \Theta(1)$

Complejidad: $\Theta\left(\sum_{i=1}^{\ell} equal(l_1[i], l_2[i])\right)$, donde $\ell = \text{mín}\{\text{long}(l_1), \text{long}(l_2)\}$. [*]

Justificaciï¿½n: [*] Ya que continua hasta que alguna de las dos listas se acabe (la de menor longitud) y en cada ciclo compara los elementos de la lista.

## Algoritmos del iterador

**iCrearIt**(**in** $l$ : `lst`) $\rightarrow res$ : iter
$res \leftarrow \langle l.primero, l\rangle$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

---

**iCrearItUlt**(**in** $l$ : `lst`) $\rightarrow res$ : iter
$res \leftarrow \langle NULL, l\rangle$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

---

**iHaySiguiente**(**in** $it$ : `iter`) $\rightarrow res : bool$
$res \leftarrow it.siguiente \neq NULL$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

---

**iHayAnterior**(**in** $it$ : `iter`) $\rightarrow res : bool$
$res \leftarrow it.siguiente \neq (it.lista \rightarrow primero)$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

**iSiguiente**(**in** $it$ : `iter`) $\rightarrow res : \alpha$
$res \leftarrow (it.siguiente \rightarrow dato)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iAnterior**(**in** $it$ : `iter`) $\rightarrow res : \alpha$
$res \leftarrow (SiguienteReal(it) \rightarrow anterior \rightarrow dato)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iAvanzar**(**in/out** $it$ : `iter`)
$it.siguiente \leftarrow (it.siguiente \rightarrow siguiente)$ ▷ $\Theta(1)$
**if** $it.siguiente = it.lista \rightarrow primero$ **then** ▷ $\Theta(1)$
    $it.siguiente \leftarrow NULL$
**end if**

Complejidad: $\Theta(1)$
Justificaciï¿½n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

**iRetroceder**(**in/out** $it$ : `iter`)
$it.siguiente \leftarrow (SiguienteReal(it) \rightarrow anterior)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

**iEliminarSiguiente**(**in/out** $it$ : `iter`)
$puntero(nodo)\ temp \leftarrow it.siguiente$
$(tmp \rightarrow siguiente \rightarrow anterior) \leftarrow (tmp \rightarrow anterior)$ ▷ Reencadenamos los nodos // $\Theta(1)$
$(tmp \rightarrow anterior \rightarrow siguiente) \leftarrow (tmp \rightarrow siguiente)$
**if** $(tmp \rightarrow siguiente) = (it.lista \rightarrow primero)$ **then** ▷ Si borramos el ï¿½ltimo nodo, ya no hay siguiente // $\Theta(1)$
    $it.siguiente \leftarrow NULL$
**else** ▷ Sino, avanzamos al siguiente // $\Theta(1)$
    $it.siguiente \leftarrow (tmp \rightarrow siguiente)$
**end if**
**if** $tmp = (it.lista \rightarrow primero)$ **then** ▷ Si borramos el primer nodo, hay que volver a setear el primero // $\Theta(1)$
    $(it.lista \rightarrow primero) \leftarrow it.siguiente$
**end if**
$tmp \leftarrow NULL$ ▷ Se libera la memoria ocupada por el nodo // $\Theta(1)$
$(it.lista \rightarrow longitud) \leftarrow (it.lista \rightarrow longitud) - 1$

Complejidad: $\Theta(1)$
Justificaciï¿½n: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

---

**iEliminarAnterior**(**in/out** $it$ : `iter`)
$Retroceder(it)$ ▷ $\Theta(1)$
$EliminarSiguiente(it)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificaciï¿½n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

**iAgregarComoSiguiente**(**in/out** $it$ : `iter`, **in** $a : \alpha$)
$AgregarComoAnterior(it, a)$ ▷ $\Theta(1)$
$Retroceder(it)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificaciï¿½n: $\Theta(1) + \Theta(1) = \Theta(1)$

---

**iAgregarComoAnterior**(**in/out** $it$ : `iter`, **in** $a : \alpha$)
$puntero(nodo)\ sig \leftarrow SiguienteReal(it)$
$puntero(nodo)\ nuevo \leftarrow$ **&** $\langle a, NULL, NULL \rangle$ ▷ Reservamos memoria para el nuevo nodo // $\Theta(1)$
**if** $sig = NULL$ **then** ▷ Asignamos los punteros de acuerdo a si el nodo es el primero o no en la lista circular // $\Theta(1)$
    $(nuevo \rightarrow anterior) \leftarrow nuevo$
    $(nuevo \rightarrow siguiente) \leftarrow nuevo$
**else**
    $(nuevo \rightarrow anterior) \leftarrow (sig \rightarrow anterior)$
    $(nuevo \rightarrow siguiente) \leftarrow sig$
**end if**
$(nuevo \rightarrow anterior \rightarrow siguiente) \leftarrow nuevo$ ▷ Reencadenamos los otros nodos // $\Theta(1)$
**if** $it.siguiente = (it.lista \rightarrow primero)$ **then** ▷ Cambiamos el primero en caso de que estemos agregando el primero // $\Theta(1)$
    $(it.lista \rightarrow primero) \leftarrow nuevo$
**end if**
$(it.lista \rightarrow longitud) \leftarrow (it.lista \rightarrow longitud) + 1$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$
Justificaciï¿½n: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

---

**iSiguienteReal**(**in** $it$ : `iter`) $\rightarrow res : puntero(nodo)$ ▷ Esta es una operaciï¿½n privada que
**if** $it.siguiente = NULL$ **then** ▷ devuelve el siguiente como lista circular // $\Theta(1)$
    $res \leftarrow (it.lista \rightarrow siguiente)$
**else**
    $res \leftarrow it.siguiente$
**end if**

Complejidad: $\Theta(1)$

---

# 5. Mï¿½dulo Pila($\alpha$)

El mï¿½dulo Pila provee una pila en la que sï¿½lo se puede acceder al tope de la misma. Por este motivo, no incluye iteradores.

Para describir la complejidad de las operaciones, vamos a llamar $copy(a)$ al costo de copiar el elemento $a \in \alpha$ (i.e., *copy* es una funciï¿½n de $\alpha$ en $\mathbb{N}$).[1]

# Interfaz

**parï¿½metros formales**
**gï¿½neros** $\alpha$
**funciï¿½n** COPIAR(**in** $a : \alpha$) $\rightarrow res : \alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} a\}$
**Complejidad:** $\Theta(copy(a))$
**Descripciï¿½n:** funciï¿½n de copia de $\alpha$'s

**se explica con**: PILA($\alpha$).

**gï¿½neros**: `pila`($\alpha$).

VACï¿½A() $\rightarrow res$ : `pila`($\alpha$)
**Pre** $\equiv$ {}
**Post** $\equiv$ $\{res =_{\text{obs}}$ vacï¿½a$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** genera una pila vacï¿½a.

APILAR(**in/out** $p$ : `pila`($\alpha$), **in** $a : \alpha$)
**Pre** $\equiv$ $\{p =_{\text{obs}} p_0\}$
**Post** $\equiv$ $\{p =_{\text{obs}} \text{apilar}(p, a)\}$
**Complejidad:** $\Theta(copy(a))$
**Descripciï¿½n:** apila $a$ en $p$
**Aliasing:** el elemento $a$ se apila por copia.

ESVACIA?(**in** $p$ : `pila`($\alpha$)) $\rightarrow res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \text{vacia?}(p)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si la pila no contiene elementos

TOPE(**in** $p$ : `pila`($\alpha$)) $\rightarrow res : \alpha$
**Pre** $\equiv$ $\{\neg$vacï¿½a?$(p)\}$
**Post** $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{tope}(p))\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el tope de la pila.
**Aliasing:** $res$ es modificable si y sï¿½lo si $p$ es modificable.

DESAPILAR(**in/out** $p$ : `pila`($\alpha$)) $\rightarrow res : \alpha$
**Pre** $\equiv$ $\{p =_{\text{obs}} p_0 \wedge \neg$vacï¿½a?$(p)\}$
**Post** $\equiv$ $\{p =_{\text{obs}} \text{desapilar}(p_0) \wedge res =_{\text{obs}} \text{tope}(p)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** desapila el tope de $p$.

TAMAï¿½O(**in** $p$ : `pila`($\alpha$)) $\rightarrow res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ tamaï¿½o$(p)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve la cantidad de elementos apilados en $p$.

[1] Nï¿½tese que este es un abuso de notaciï¿½n, ya que no estamos describiendo *copy* en funciï¿½n del tamaï¿½o de $a$. A la hora de usarlo, habrï¿½ que realizar la traducciï¿½n

Copiar(**in** $p$ : `pila`($\alpha$)) $\rightarrow$ *res* : `pila`($\alpha$)
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} p\}$
**Complejidad:** $\Theta\left(\sum_{i=1}^{t} copy(p[i])\right) = O\left(t \max_{i=1}^{t} copy(p[i])\right)$, donde $t$ = tamaï¿½o($p$).
**Descripciï¿½n:** genera una copia nueva de la pila

$\bullet = \bullet$(**in** $p_1$ : `pila`($\alpha$), **in** $p_2$ : `pila`($\alpha$)) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} p_1 = p_2\}$
**Complejidad:** $\Theta\left(\sum_{i=1}^{t} equal(p_1[i], p_2[i])\right)$, donde $t$ = mín{tamaï¿½o($p_1$), tamaï¿½o($p_2$)}.
**Descripciï¿½n:** compara $p_1$ y $p_2$ por igualdad, cuando $\alpha$ posee operaciï¿½n de igualdad.
**Requiere:** $\bullet = \bullet$(**in** $a_1$ : $\alpha$, **in** $a_2$ : $\alpha$) $\rightarrow$ *res* : `bool`
  **Pre** $\equiv$ {true}
  **Post** $\equiv$ $\{res =_{\text{obs}} (a_1 = a_2)\}$
  **Complejidad:** $\Theta(equal(a_1, a_2))$
  **Descripciï¿½n:** funciï¿½n de igualdad de $\alpha$'s

### Especificaciï¿½n de las operaciones auxiliares utilizadas en la interfaz

**TAD** Pila Extendida($\alpha$)

**extiende** Pila($\alpha$)

**otras operaciones (no exportadas)**

$\bullet[\bullet]$ : pila($\alpha$) $p$ $\times$ nat $i$ $\longrightarrow$ $\alpha$   {$i$ < tamaï¿½o($p$)}

**axiomas**

$p[i]$ $\equiv$ **if** $i = 0$ **then** tope($p$) **else** desapilar($p$)[$i - 1$] **fi**

**Fin TAD**

## Representaciï¿½n

El objetivo de este mï¿½dulo es implementar una pila lo mï¿½s eficientemente posible, y eso se puede obtener utilizando una lista enlazada. Claramente, cualquier lista representa una pila, donde el tope se encuentra o en el primer o en el ï¿½ltimo elemento. En este caso, elegimos que el tope se encuentre en el primer elemento.

`pila`($\alpha$) **se representa con** `lista`($\alpha$)

Rep : lista($\alpha$) $\longrightarrow$ bool
Rep($l$) $\equiv$ true

Abs : lista($\alpha$) $l$ $\longrightarrow$ pila($\alpha$)   {Rep($l$)}
Abs($l$) $\equiv$ **if** vacia?($l$) **then** vacï¿½a **else** apilar(prim($l$), Abs(fin($l$))) **fi**

## Algoritmos

# 6. Mï¿½dulo Cola($\alpha$)

El mï¿½dulo Cola provee una cola en la que sï¿½lo se puede acceder al proximo de la misma. Por este motivo, no incluye iteradores.

Para describir la complejidad de las operaciones, vamos a llamar *copy*($a$) al costo de copiar el elemento $a \in \alpha$ (i.e., *copy* es una funciï¿½n de $\alpha$ en $\mathbb{N}$).[2]

[2] Nï¿½tese que este es un abuso de notaciï¿½n, ya que no estamos describiendo *copy* en funciï¿½n del tamaï¿½o de $a$. A la hora de usarlo, habrï¿½ que realizar la traducciï¿½n

# Interfaz

**parï¿½metros formales**
**gï¿½neros** $\alpha$
**funciï¿½n** Copiar(**in** $a : \alpha$) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} a\}$
**Complejidad:** $\Theta(copy(a))$
**Descripciï¿½n:** funciï¿½n de copia de $\alpha$'s

**se explica con**: Cola($\alpha$).

**gï¿½neros**: `cola`($\alpha$).

Vacï¿½a() $\rightarrow$ $res$ : `cola`($\alpha$)
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ vacï¿½a$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** genera una cola vacï¿½a.

Encolar(**in/out** $c$: `cola`($\alpha$), **in** $a : \alpha$)
**Pre** $\equiv$ $\{c =_{\text{obs}} c_0\}$
**Post** $\equiv$ $\{p =_{\text{obs}}$ encolar$(c, a)\}$
**Complejidad:** $\Theta(copy(a))$
**Descripciï¿½n:** encola $a$ a $c$
**Aliasing:** el elemento $a$ se encola por copia.

EsVacia?(**in** $c$: `cola`($\alpha$)) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ vacia?$(c)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si la cola es vacï¿½a.

Proximo(**in** $c$: `cola`($\alpha$)) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {$\not{v}$acï¿½a?$(c)$}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ proximo$(c)$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el proximo de la cola.
**Aliasing:** $res$ es modificable si y sï¿½lo si $p$ es modificable.

Desencolar(**in/out** $c$: `cola`($\alpha$))
**Pre** $\equiv$ $\{c =_{\text{obs}} c_0 \wedge \neg$vacï¿½a?$(c)\}$
**Post** $\equiv$ $\{c =_{\text{obs}}$ desacolar$(c_0)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** desencola el proximo de $c$.

Tamaï¿½o(**in** $c$: `cola`($\alpha$)) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ tamaï¿½o$(c)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve la cantidad de elementos encolados en $c$.

Copiar(**in** $c$: `cola`($\alpha$)) $\rightarrow$ $res$ : `cola`($\alpha$)
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} c\}$
**Complejidad:** $\Theta\left(\sum_{i=1}^{t} copy(c[i])\right)$, donde $t =$ tamaï¿½o$(c)$
**Descripciï¿½n:** genera una copia nueva de la cola

$\bullet = \bullet$(**in** $c_1$ : `cola`($\alpha$), **in** $c_2$ : `cola`($\alpha$)) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}

**Post** $\equiv \{res =_{\text{obs}} c_1 = c_2\}$
**Complejidad:** $\Theta\left(\sum_{i=1}^{t} equal(c_1[i], c_2[i])\right)$, donde $t = \text{mín}\{\text{tamaï¿½o}(c_1), \text{tamaï¿½o}(c_2)\}$.
**Descripciï¿½n:** compara $c_1$ y $c_2$ por igualdad, cuando $\alpha$ posee operaciï¿½n de igualdad.
**Requiere:** $\bullet = \bullet$(**in** $a_1 : \alpha$, **in** $a_2 : \alpha$) $\to res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv \{res =_{\text{obs}} (a_1 = a_2)\}$
**Complejidad:** $\Theta(equal(a_1, a_2))$
**Descripciï¿½n:** funciï¿½n de igualdad de $\alpha$'s

### Especificaciï¿½n de las operaciones auxiliares utilizadas en la interfaz

**TAD** Cola Extendida($\alpha$)

**extiende** COLA($\alpha$)

**otras operaciones (no exportadas)**

$\bullet[\bullet]$ : cola($\alpha$) $p$ × nat $i$ $\longrightarrow$ $\alpha$ $\{i < \text{tamaï¿½o}(p)\}$

**axiomas**

$c[i]$ $\equiv$ **if** $i = 0$ **then** proximo($c$) **else** desencolar($c$)[$i - 1$] **fi**

**Fin TAD**

## Representaciï¿½n

El objetivo de este mï¿½dulo es implementar una cola lo mï¿½s eficientemente posible, y eso se puede obtener utilizando una lista enlazada. Claramente, cualquier lista representa una cola, donde el proximo se encuentra o en el primer o en el ï¿½ltimo elemento. En este caso, elegimos que el proximo se encuentre en el primer elemento.

`cola`($\alpha$) **se representa con** `lista`($\alpha$)

Rep : lista($\alpha$) $\longrightarrow$ bool
Rep($l$) $\equiv$ true

Abs : lista($\alpha$) $l$ $\longrightarrow$ cola($\alpha$) {Rep($l$)}
Abs($l$) $\equiv$ **if** vacia?($l$) **then** vacï¿½a **else** encolar(ult($l$), Abs(com($l$))) **fi**

## Algoritmos

# 7. Mï¿½dulo Vector($\alpha$)

El mï¿½dulo Vector provee una secuencia que permite obtener el $i$-ï¿½simo elemento de forma eficiente. La inserciï¿½n de elementos es eficiente cuando se realiza al final de la misma, si se utiliza un anï¿½lisis amortizado (i.e., $n$ inserciones consecutivas cuestan $O(n)$), aunque puede tener un costo lineal en peor caso. La inserciï¿½n en otras posiciones no es tan eficiente, ya que requiere varias copias de elementos. El borrado de los ï¿½ltimos elementos es eficiente, no asï¿½ el borrado de los elementos intermedios.

Una consideraciï¿½n a tener en cuenta, es que el espacio utilizado por la estructura es el mï¿½ximo espacio utilizado en cualquier momento del programa. Es decir, si se realizan $n$ inserciones seguidas de $n$ borrados, el espacio utilizado es $O(n)$ por el espacio de cada $\alpha$. Si fuera necesario borrar esta memoria, se puede crear una copia del vector con los elementos sobrevivientes, borrando la copia vieja.

En cuanto al recorrido de los elementos, como los mismos se pueden recorrer con un ï¿½ndice, no se proveen iteradores.

Para describir la complejidad de las operaciones, vamos a llamar *copy*($a$) al costo de copiar el elemento $a \in \alpha$ (i.e., *copy* es una funciï¿½n de $\alpha$ en $\mathbb{N}$), y vamos a utilizar

$$f(n) = \begin{cases} n & \text{si } n = 2^k \text{ para algï¿½n } k \\ 1 & \text{en caso contrario} \end{cases}$$

para describir el costo de inserciï¿½n de un elemento. Vale la pena notar que $\sum_{i=1}^{n} \frac{f(j+i)}{n} \to 1$ cuando $n \to \infty$, para todo $j \in \mathbb{N}$. En otras palabras, la inserciï¿½n consecutiva de $n$ elementos costarï¿½ $O(1)$ copias por elemento, en tï¿½rminos asintï¿½ticos.

# Interfaz

**parï¿½metros formales**
**gï¿½neros** $\alpha$
**funciï¿½n** Copiar(**in** $a : \alpha$) $\to$ $res$ : $\alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} a\}$
**Complejidad:** $\Theta(copy(a))$
**Descripciï¿½n:** funciï¿½n de copia de $\alpha$'s.

**se explica con**: Secu($\alpha$).

**gï¿½neros**: `vector`($\alpha$).

Vacï¿½a() $\to$ $res$ : `vector`($\alpha$)
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} <>\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** genera un vector vacï¿½o.

AgregarAtras(**in/out** $v$: `vector`($\alpha$), **in** $a : \alpha$)
**Pre** $\equiv$ $\{v =_{\text{obs}} v_0\}$
**Post** $\equiv$ $\{v =_{\text{obs}} v_0 \circ a\}$
**Complejidad:** $\Theta(f(\text{long}(v)) + copy(a))$
**Descripciï¿½n:** agrega el elemento $a$ como ï¿½ltimo elemento del vector.
**Aliasing:** el elemento $a$ se agrega por copia. Cualquier referencia que se tuviera al vector queda invalidada cuando long($v$) es potencia de 2.

EsVacï¿½o?(**in** $v$: `vector`($\alpha$)) $\to$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \text{vacia?}(v)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si $v$ esta vacï¿½o

Comienzo(**in/out** $v$: `vector`($\alpha$))
**Pre** $\equiv$ $\{v =_{\text{obs}} v_0 \land \neg\text{vacï¿½a?}(v)\}$
**Post** $\equiv$ $\{v =_{\text{obs}} \text{com}(v_0)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** elimina el ï¿½ltimo elemento de $v$.

TomarPrimeros(**in/out** $v$: `vector`($\alpha$), **in** $n$: `nat`)
**Pre** $\equiv$ $\{v =_{\text{obs}} v_0\}$
**Post** $\equiv$ $\{v =_{\text{obs}} \text{Tomar}(v_0, n)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** elimina los ï¿½ltimos máx{long($v$) - $n$, 0} elementos del vector, i.e., se queda con los primeros $n$ elementos del vector.

TirarUltimos(**in/out** $v$: `vector`($\alpha$), **in** $n$: `nat`)
**Pre** $\equiv$ $\{v =_{\text{obs}} v_0\}$
**Post** $\equiv$ $\{v =_{\text{obs}} \text{Tomar}(v_0, \text{long}(v_0) - n)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** elimina los ï¿½ltimos máx{long($v$), $n$} elementos del vector.

Ultimo(**in** $v$: `vector`($\alpha$)) $\to$ $res$ : $\alpha$
**Pre** $\equiv$ $\{\neg\text{vacï¿½a?}(v)\}$
**Post** $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{ult}(v))\}$

**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el ï¿½ltimo elemento del vector.
**Aliasing:** $res$ es modificable si y sï¿½lo si $v$ es modificable.

LONGITUD(**in** $l$: `vector(`$\alpha$`)`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\mathrm{obs}} \mathrm{long}(v)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve la cantidad de elementos que contiene el vector.

$\bullet[\bullet]$(**in** $v$: `vector(`$\alpha$`)`, **in** $i$: `nat`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ $\{i < \mathrm{long}(v)\}$
**Post** $\equiv$ $\{\mathrm{alias}(res =_{\mathrm{obs}} \mathrm{iesimo}(v,\ i))\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento que se encuentra en la $i$-ï¿½sima posiciï¿½n del vector en base 0. Es decir, $v[i]$ devuelve el elemento que se encuentra en la posiciï¿½n $i+1$.
**Aliasing:** $res$ es modificable si y sï¿½lo si $v$ es modificable.

AGREGAR(**in/out** $v$: `vector(`$\alpha$`)`, **in** $i$: `nat`, **in** $a$: $\alpha$)
**Pre** $\equiv$ $\{v =_{\mathrm{obs}} v_0 \wedge i \leq \mathrm{long}(v)\}$
**Post** $\equiv$ $\{v =_{\mathrm{obs}} \mathrm{Agregar}(v,\ i,\ a)\}$
**Complejidad:** $\Theta(f(\mathrm{long}(v)) + \mathrm{long}(v) - i + \mathit{copy}(a))$
**Descripciï¿½n:** agrega el elemento $a$ a $v$, de forma tal que ocupe la posiciï¿½n $i$.
**Aliasing:** el elemento $a$ se agrega por copia. Cualquier referencia que se tuviera al vector queda invalidada cuando $\mathrm{long}(v)$ es potencia de 2.

ELIMINAR(**in/out** $v$: `vector(`$\alpha$`)`, **in** $i$: `nat`)
**Pre** $\equiv$ $\{v =_{\mathrm{obs}} v_0 \wedge i < \mathrm{long}(v)\}$
**Post** $\equiv$ $\{v =_{\mathrm{obs}} \mathrm{Eliminar}(v,\ i)\}$
**Complejidad:** $\Theta(\mathrm{long}(v) - i)$
**Descripciï¿½n:** elimina el elemento que ocupa la posiciï¿½n $i$ de $v$.

COPIAR(**in** $v$: `vector(`$\alpha$`)`) $\rightarrow$ $res$ : `vector(`$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\mathrm{obs}} v\}$
**Complejidad:** $\Theta\left(\sum_{i=1}^{\ell} \mathit{copy}(v[i])\right)$, donde $\ell = \mathrm{long}(v)$.
**Descripciï¿½n:** genera una copia nueva del vector.

$\bullet = \bullet$(**in** $v_1$: `vector(`$\alpha$`)`, **in** $v_2$: `vector(`$\alpha$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\mathrm{obs}} v_1 = v_2\}$
**Complejidad:** $\Theta\left(\sum_{i=1}^{\ell} \mathit{equal}(v_1[i], v_2[i])\right)$, donde $\ell = \min\{\mathrm{long}(v_1), \mathrm{long}(v_2)\}$.
**Descripciï¿½n:** compara $v_1$ y $v_2$ por igualdad, cuando $\alpha$ posee operaciï¿½n de igualdad.
**Requiere:** $\bullet = \bullet$(**in** $a_1$: $\alpha$, **in** $a_2$: $\alpha$) $\rightarrow$ $res$ : `bool`
  **Pre** $\equiv$ {true}
  **Post** $\equiv$ $\{res =_{\mathrm{obs}} (a_1 = a_2)\}$
  **Complejidad:** $\Theta(\mathit{equal}(a_1, a_2))$
  **Descripciï¿½n:** funciï¿½n de igualdad de $\alpha$'s

## Especificaciï¿½n de las operaciones auxiliares utilizadas en la interfaz

**TAD** Secuencia Extendida($\alpha$)

**extiende** SECUENCIA($\alpha$)

**otras operaciones (exportadas)**

| | | |
|---|---|---|
| Agregar | : secu($\alpha$) $s$ × nat $i$ × $\alpha$ $a$ $\longrightarrow$ secu($\alpha$) | $\{i \leq \mathrm{long}(s)\}$ |
| Eliminar | : secu($\alpha$) $s$ × nat $i$ $\longrightarrow$ secu($\alpha$) | $\{i < \mathrm{long}(s)\}$ |

Tomar : secu($\alpha$) × nat ⟶ secu($\alpha$)

**otras operaciones (no exportadas)**

Tirar : secu($\alpha$) × nat ⟶ secu($\alpha$)

**axiomas**

Agregar$(s, i, a)$ ≡ (Tomar($n$, $i$) ∘ $a$) & Tirar($n$, $i$)

Eliminar$(s, i, a)$ ≡ (Tomar($n$, $i-1$) & Tirar($n$, $i$)

Tomar$(s, n)$ ≡ **if** $n = 0 \lor$ vacia?($s$) **then** <> **else** prim($s$) • Tomar(fin($s$), $n-1$) **fi**

Tirar$(s, n)$ ≡ **if** $n = 0 \lor$ vacia?($s$) **then** $s$ **else** Tirar(fin($s$), $n-1$) **fi**

**Fin TAD**

# Representaciï¿½n

La idea de este mï¿½dulo es tener una lista donde el $i$-ï¿½simo se puede obtener en tiempo $O(1)$. Para esto, necesitamos usar algï¿½n tipo de acceso aleatorio a los elementos, que se consigue utilizando un arreglo. Ademas, necesitamos que el agregado de elementos tome $O(1)$ copias cuando analizamos el tiempo amortizado, i.e., $O(f(n))$ copias. Para lograr esto, podemos duplicar el tamaï¿½o del arreglo cuando este se llena.

`vector`($\alpha$) **se representa con** `vec`

donde `vec` es `tupla`(*elementos*: `arreglo_dimensionable de puntero`($\alpha$), *longitud*: `nat`)

Rep : vec ⟶ bool

Rep($v$) ≡ true ⟺ ($\exists k$: nat)(tam($v$.elementos) = $2^k$ ∧ $v$.longitud ≤ tam($v$.elementos) ∧
($\forall i$: nat)($0 \le i < v$.longitud $\Rightarrow_L$ def?($v$.elementos, $i$)) ∧
($\forall i, j$: nat)($0 \le i < j < v$.longitud $\Rightarrow_L$ $v$.elementos[$i$] ≠ $v$.elementos[$j$]))

Abs : vec $v$ ⟶ secu($\alpha$) {Rep($v$)}

Abs($v$) ≡ **if** $v$.longitud = 0 **then**
<>
**else**
Abs(⟨$v$.elementos, $v$.longitud − 1⟩) ∘ ∗($v$.elementos[$v$.longitud − 1])
**fi**

# Algoritmos

# 8. Mï¿½dulo Diccionario Lineal($\kappa$, $\sigma$)

El mï¿½dulo Diccionario Lineal provee un diccionario bï¿½sico en el que se puede definir, borrar, y testear si una clave estï¿½ definida en tiempo lineal. Cuando ya se sabe que la clave a definir no esta definida en el diccionario, la definiciï¿½n se puede hacer en tiempo $O(1)$.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite recorrer y eliminar los elementos de $d$ como si fuera una secuencia de pares $\kappa, \sigma$.

Para describir la complejidad de las operaciones, vamos a llamar *copy*($k$) al costo de copiar el elemento $k \in \kappa \cup \sigma$ y *equal*($k_1, k_2$) al costo de evaluar si dos elementos $k_1, k_2 \in \kappa$ son iguales (i.e., *copy* y *equal* son funciones de $\kappa \cup \sigma$ y $\kappa \times \kappa$ en $\mathbb{N}$, respectivamente).[3]

# Interfaz

**parï¿½metros formales**

**gï¿½neros** $\kappa, \sigma$

**funciï¿½n** • = •(**in** $k_1$ : $\kappa$, **in** $k_2$ : $\kappa$) → *res* : `bool`

**Pre** ≡ {true}

**Post** ≡ {$res =_{\text{obs}} (k_1 = k_2)$}

**Complejidad:** $\Theta(equal(k_1, k_2))$

**Descripciï¿½n:** funciï¿½n de igualdad de $\kappa$'s

**funciï¿½n** Copiar(**in** $k$ : $\kappa$) → *res* : $\kappa$

**Pre** ≡ {true}

**Post** ≡ {$res =_{\text{obs}} k$}

**Complejidad:** $\Theta(copy(k))$

**Descripciï¿½n:** funciï¿½n de copia de $\kappa$'s

[3] Nï¿½tese que este es un abuso de notaciï¿½n, ya que no estamos describiendo *copy* y *equal* en funciï¿½n del tamaï¿½o de $k$. A la hora de usarlo, habrï¿½ que realizar la traducciï¿½n.

**funciï¿½n** COPIAR(**in** $s : \sigma$) $\to res : \sigma$
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} s\}$
**Complejidad:** $\Theta(copy(s))$
**Descripciï¿½n:** funciï¿½n de copia de $\sigma$'s

**se explica con**: DICCIONARIO($\kappa, \sigma$), ITERADOR BIDIRECCIONAL(TUPLA($\kappa, \sigma$)).

**gï¿½neros**: `dicc`($\kappa, \sigma$), `itDicc`($\kappa, \sigma$).

## Operaciones bï¿½sicas de diccionario

VACï¿½O() $\to res$ : `dicc`($\kappa, \sigma$)
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ vacio$\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** genera un diccionario vacï¿½o.

DEFINIR(**in/out** $d$: `dicc`($\kappa, \sigma$), **in** $k : \kappa$, **in** $s : \sigma$) $\to res$ : `itDicc`($\kappa, \sigma$)
**Pre** $\equiv$ $\{d =_{\text{obs}} d_0\}$
**Post** $\equiv$ $\{d =_{\text{obs}}$ definir$(d, k, s) \wedge$ haySiguiente$(res) \wedge_L$ Siguiente$(res) = \langle k, s\rangle \wedge$ alias(esPermutaciï¿½n(SecuSuby($res$), $d$))$\}$
**Complejidad:** $\Theta\left(\sum_{k' \in K} equal(k, k') + copy(k) + copy(s)\right)$, donde $K$ = claves($d$)
**Descripciï¿½n:** define la clave $k$ con el significado $s$ en el diccionario. Retorna un iterador al elemento reciï¿½n agregado.
**Aliasing:** los elementos $k$ y $s$ se definen por copia. El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n ELIMINARSIGUIENTE. Ademï¿½s, anteriores($res$) y siguientes($res$) podrï¿½an cambiar completamente ante cualquier operaciï¿½n que modifique el $d$ sin utilizar las funciones del iterador.

DEFINIRRAPIDO(**in/out** $d$: `dicc`($\kappa, \sigma$), **in** $k : \kappa$, **in** $s : \sigma$) $\to res$ : `itDicc`($\kappa, \sigma$)
**Pre** $\equiv$ $\{d =_{\text{obs}} d_0 \wedge \neg$definido?$(d,\ k)\}$
**Post** $\equiv$ $\{d =_{\text{obs}}$ definir$(d, k, s) \wedge$ haySiguiente$(res) \wedge_L$ Siguiente$(res) = \langle k, s\rangle \wedge$ esPermutaciï¿½n(SecuSuby($res$), $d$)$\}$
**Complejidad:** $\Theta(copy(k) + copy(s))$
**Descripciï¿½n:** define la clave $k \notin$ claves($d$) con el significado $s$ en el diccionario. Retorna un iterador al elemento reciï¿½n agregado.
**Aliasing:** los elementos $k$ y $s$ se definen por copia. El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n ELIMINARSIGUIENTE. Ademï¿½s, anteriores($res$) y siguientes($res$) podrï¿½an cambiar completamente ante cualquier operaciï¿½n que modifique el $d$ sin utilizar las funciones del iterador.

DEFINIDO?(**in** $d$: `dicc`($\kappa, \sigma$), **in** $k : \kappa$) $\to res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}}$ def?$(d,\ k)\}$
**Complejidad:** $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K$ = claves($d$)
**Descripciï¿½n:** devuelve `true` si y sï¿½lo $k$ estï¿½ definido en el diccionario.

SIGNIFICADO(**in** $d$: `dicc`($\kappa, \sigma$), **in** $k : \kappa$) $\to res : \sigma$
**Pre** $\equiv$ {def?$(d,\ k)$}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Significado$(d,\ k)$)}
**Complejidad:** $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K$ = claves($d$)
**Descripciï¿½n:** devuelve el significado de la clave $k$ en $d$.
**Aliasing:** $res$ es modificable si y sï¿½lo si $d$ es modificable.

BORRAR(**in/out** $d$: `dicc`($\kappa, \sigma$), **in** $k : \kappa$)
**Pre** $\equiv$ $\{d = d_0 \wedge$ def?$(d,\ k)\}$
**Post** $\equiv$ $\{d =_{\text{obs}}$ borrar$(d_0, k)\}$

**Complejidad:** $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K$ = claves($d$)
**Descripciï¿½n:** elimina la clave $k$ y su significado de $d$.

#Claves(**in** $d$: `dicc(`$\kappa$, $\sigma$`)`) $\rightarrow$ *res* : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \#\text{claves}(d)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve la cantidad de claves del diccionario.

Copiar(**in** $d$: `dicc(`$\kappa$, $\sigma$`)`) $\rightarrow$ *res* : `dicc(`$\kappa$, $\sigma$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} d\}$
**Complejidad:** $\Theta\left(\sum_{k \in K} (copy(k) + copy(\text{significado}(k, d)))\right)$, donde $K$ = claves($d$)
**Descripciï¿½n:** genera una copia nueva del diccionario.

$\bullet = \bullet$(**in** $d_1$ : `dicc(`$\kappa$, $\sigma$`)`, **in** $d_2$ : `dicc(`$\kappa$, $\sigma$`)`) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} c_1 = c_2\}$
**Complejidad:** $O\left(\sum_{\substack{k_1 \in K_1 \\ k_2 \in K_2}} equal(\langle k_1, s_1\rangle, \langle k_2, s_2\rangle)\right)$, donde $K_i$ = claves($d_i$) y $s_i$ = significado($d_i, k_i$), $i \in \{1, 2\}$.
**Descripciï¿½n:** compara $d_1$ y $d_2$ por igualdad, cuando $\sigma$ posee operaciï¿½n de igualdad.
**Requiere:** $\bullet = \bullet$(**in** $s_1 : \sigma$, **in** $s_2 : \sigma$) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} (s_1 = s_2)\}$
**Complejidad:** $\Theta(equal(s_1, s_2))$
**Descripciï¿½n:** funciï¿½n de igualdad de $\sigma$'s

## Operaciones del iterador

El iterador que presentamos permite modificar el diccionario recorrido, eliminando elementos. Sin embargo, cuando el diccionario es no modificable, no se pueden utilizar las funciones de eliminaciï¿½n. Ademï¿½s, las claves de los elementos iterados no pueden modificarse nunca, por cuestiones de implementaciï¿½n. Cuando $d$ es modificable, decimos que *it* es modificable.

Para simplificar la notaciï¿½n, vamos a utilizar clave y significado en lugar de $\Pi_1$ y $\Pi_2$ cuando utilicemos una tupla($\kappa, \sigma$).

CrearIt(**in** $d$: `dicc(`$\kappa$, $\sigma$`)`) $\rightarrow$ *res* : `itDicc(`$\kappa$, $\sigma$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {alias(esPermutaciï¿½n(SecuSuby(*res*), $d$)) $\wedge$ vacia?(Anteriores(*res*))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** crea un iterador bidireccional del diccionario, de forma tal que HayAnterior evalï¿½e a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).
**Aliasing:** El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n EliminarSiguiente. Ademï¿½s, anteriores(*res*) y siguientes(*res*) podrï¿½an cambiar completamente ante cualquier operaciï¿½n que modifique $d$ sin utilizar las funciones del iterador.

HaySiguiente(**in** *it*: `itDicc(`$\kappa$, $\sigma$`)`) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \text{haySiguiente?}(it)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si en el iterador todavï¿½a quedan elementos para avanzar.

HayAnterior(**in** *it*: `itDicc(`$\kappa$, $\sigma$`)`) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \text{hayAnterior?}(it)\}$
**Complejidad:** $\Theta(1)$

**Descripciï¿½n:** devuelve `true` si y sï¿½lo si en el iterador todavï¿½a quedan elementos para retroceder.

SIGUIENTE(**in** $it$: `itDicc(`$\kappa, \sigma$`)`) $\rightarrow$ $res$ : `tupla(`$\kappa, \sigma$`)`
**Pre** $\equiv$ {HaySiguiente?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Siguiente($it$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento siguiente del iterador.
**Aliasing:** $res$.significado es modificable si y sï¿½lo si $it$ es modificable. En cambio, $res$.clave no es modificable.

SIGUIENTECLAVE(**in** $it$: `itDicc(`$\kappa, \sigma$`)`) $\rightarrow$ $res$ : $\kappa$
**Pre** $\equiv$ {HaySiguiente?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Siguiente($it$).clave)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve la clave del elemento siguiente del iterador.
**Aliasing:** $res$ no es modficable.

SIGUIENTESIGNIFICADO(**in** $it$: `itDicc(`$\kappa, \sigma$`)`) $\rightarrow$ $res$ : $\sigma$
**Pre** $\equiv$ {HaySiguiente?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Siguiente($it$).significado)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el significado del elemento siguiente del iterador.
**Aliasing:** $res$ es modificable si y sï¿½lo si $it$ es modficable.

ANTERIOR(**in** $it$: `itDicc(`$\kappa, \sigma$`)`) $\rightarrow$ $res$ : `tupla(clave: `$\kappa$`, significado: `$\sigma$`)`
**Pre** $\equiv$ {HayAnterior?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Anterior($it$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento anterior del iterador.
**Aliasing:** $res$.`significado` es modificable si y sï¿½lo si $it$ es modificable. En cambio, $res$.`clave` no es modificable.

ANTERIORCLAVE(**in** $it$: `itDicc(`$\kappa, \sigma$`)`) $\rightarrow$ $res$ : $\kappa$
**Pre** $\equiv$ {HayAnterior?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Anterior($it$).clave)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve la clave del elemento anterior del iterador.
**Aliasing:** $res$ no es modficable.

ANTERIORSIGNIFICADO(**in** $it$: `itDicc(`$\kappa, \sigma$`)`) $\rightarrow$ $res$ : $\sigma$
**Pre** $\equiv$ {HayAnterior?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Anterior($it$).significado)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el significado del elemento anterior del iterador.
**Aliasing:** $res$ es modificable si y sï¿½lo si $it$ es modficable.

AVANZAR(**in/out** $it$: `itDicc(`$\kappa, \sigma$`)`)
**Pre** $\equiv$ {$it = it_0 \land$ HaySiguiente?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ Avanzar($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** avanza a la posiciï¿½n siguiente del iterador.

RETROCEDER(**in/out** $it$: `itDicc(`$\kappa, \sigma$`)`)
**Pre** $\equiv$ {$it = it_0 \land$ HayAnterior?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ Retroceder($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** retrocede a la posiciï¿½n anterior del iterador.

ELIMINARSIGUIENTE(**in/out** $it$: `itDicc(`$\kappa, \sigma$`)`)
**Pre** $\equiv$ {$it = it_0 \land$ HaySiguiente?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ EliminarSiguiente($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** elimina del diccionario la clave del elemento que se encuentra en la posiciï¿½n siguiente.

EliminarAnterior(**in/out** $it$: `itDicc(`$\kappa, \sigma$`)`)
**Pre** $\equiv \{it = it_0 \wedge \text{HayAnterior?}(it)\}$
**Post** $\equiv \{it =_{\text{obs}} \text{EliminarAnterior}(it_0)\}$
**Complejidad:** $\Theta(1)$
**Descripcii¿½n:** elimina del diccionario la clave del elemento que se encuentra en la posicii¿½n anterior.

## Especificacii¿½n de las operaciones auxiliares utilizadas en la interfaz

**TAD** Diccionario Extendido($\kappa$, $\sigma$)

**extiende** Diccionario($\kappa, \sigma$)

**otras operaciones (no exportadas)**

esPermutacion? : secu(tupla($\kappa, \sigma$)) $\times$ dicc($\kappa, \sigma$) $\longrightarrow$ bool
secuADicc : secu(tupla($\kappa, \sigma$)) $\longrightarrow$ dicc($\kappa, \sigma$)

**axiomas**

esPermutacion?($s, d$) $\equiv$ $d$ = secuADicc($s$) $\wedge$ #claves($d$) = long($s$)
secuADicc($s$) $\equiv$ **if** vacia?($s$) **then** vacio **else** definir($\Pi_1$(prim($s$)), $\Pi_2$(prim($s$)), secuADict(fin($s$))) **fi**

**Fin TAD**

# Representacii¿½n

## Representacii¿½n del diccionario

Hay dos opciones bi¿½sicas para representar el diccionario lineal, con sus pros y sus contras. La que parece mi¿½s natural, es representarlo como un conjunto de tuplas sobre secuencia (ver Seccion **??**). La ventaja de esta representacii¿½n es que el invariante de representacii¿½n y la funcii¿½n de abstraccii¿½n resultan un poco mi¿½s naturales. La desventaja es que, como en un conjunto no se pueden modificar los valores, no podri¿½amos modificar el significado de una clave dada. Esto es contrario a lo que queremos. Una opcii¿½n alternativa por este camino, es definir el diccionario como un conjunto de claves y conjunto de significados, donde cada clave guarda un iterador o puntero a un significado. Esta opcii¿½n puede resultar viable, pero es un poco molesta.

La representacii¿½n que optamos consiste en definir al diccionario como dos listas, una de claves y otra de significados. La lista de claves no puede tener repetidos, mientras que la de significados si puede. Ademas, la $i$-i¿½sima clave de la lista se asocia al $i$-i¿½simo significado. En cierto sentido, estamos definiendo al diccionario como un conjunto de claves y una secuencia de significados. Para no repetir la representacii¿½n y el codigo del diccionario en el conjunto, vamos a representar al conjunto como un diccionario (ver Seccii¿½n **??**). Si bien esto no parece ser una solucii¿½n natural, tampoco es tan rara, y nos permite resolver el problema reutilizando la mayori¿½a del codigo.

`dicc(`$\kappa, \sigma$`)` **se representa con** `dic`

donde `dic` es `tupla(`*claves*: `lista(`$\kappa$`)`, *significados*: `lista(`$\sigma$`)`

Rep : dic $\longrightarrow$ bool
Rep($d$) $\equiv$ true $\Longleftrightarrow$ #claves(secuADicc($d$.claves)) = long($d$.claves) $\wedge$ long($d$.claves) = long($d$.significados)

Abs : dicc $d$ $\longrightarrow$ dicc($\kappa, \sigma$) {Rep($d$)}
Abs($d$) $\equiv$ **if** vaci¿½a?($d$.claves) **then** vaci¿½o **else** definir(prim($d$).claves, prim($d$).significado, Abs(fin($d$))) **fi**

## Representacii¿½n del iterador

El iterador del diccionario es simplemente un par de iteradores a las listas correspondientes. Lo i¿½nico que hay que pedir es que se satisfaga el Rep de este par de listas.

`itDicc(`$\kappa$`,` $\sigma$`)` **se representa con** `itDic`

donde `itDic` es `tupla(`*claves*: `itLista(`$\kappa$`)`, *significados*: `itLista(`$\sigma$`))`

Rep : itDic $\longrightarrow$ bool
Rep($it$) $\equiv$ true $\Longleftrightarrow$ Rep($\langle$SecuSuby($it$.claves), SecuSuby($it$.significados)$\rangle$)

Abs : itDic $it$ $\longrightarrow$ itBi(tupla($\kappa$, $\sigma$)) {Rep($it$)}
Abs($it$) $\equiv$ CrearItBi(Join(Anteriores($it$.claves), Anteriores($it$.significados)),
Join(Siguientes($it$.claves), Siguientes($it$.significados)))

Join : secu($\alpha$) $a$ $\times$ secu($\beta$) $b$ $\longrightarrow$ secu(tupla($\alpha, \beta$)) {long($a$) = long($b$)}
Join($a, b$) $\equiv$ **if** vacia?($a$) **then** <> **else** $\langle$prim($a$), prim($b$)$\rangle$ $\bullet$ Join(Fin($a$), Fin($b$)) **fi**

# Algoritmos

## 9. Mï¿½dulo Conjunto Lineal($\alpha$)

El mï¿½dulo Conjunto Lineal provee un conjunto bï¿½sico en el que se puede insertar, eliminar, y testear pertenencia en tiempo lineal (de comparaciones y/o copias). Cuando ya se sabe que el elemento a insertar no pertenece al conjunto, la inserciï¿½n se puede hacer con complejidad de $O(1)$ copias.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que permite eliminar los elementos iterados.

Para describir la complejidad de las operaciones, vamos a llamar *copy*($a$) al costo de copiar el elemento $a \in \alpha$ y *equal*($a_1, a_2$) al costo de evaluar si dos elementos $a_1, a_2 \in \alpha$ son iguales (i.e., *copy* y *equal* son funciones de $\alpha$ y $\alpha \times \alpha$ en $\mathbb{N}$, respectivamente).[4]

# Interfaz

**parï¿½metros formales**
**gï¿½neros** $\alpha$
**funciï¿½n** $\bullet = \bullet$(**in** $a_1 : \alpha$, **in** $a_2 : \alpha$) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} (a_1 = a_2)\}$
**Complejidad:** $\Theta(equal(a_1, a_2))$
**Descripciï¿½n:** funciï¿½n de igualdad de $\alpha$'s

**funciï¿½n** Copiar(**in** $a : \alpha$) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} a\}$
**Complejidad:** $\Theta(copy(a))$
**Descripciï¿½n:** funciï¿½n de copia de $\alpha$'s

**se explica con**: Conj($\alpha$), Iterador Bidireccional Modificable($\alpha$).

**gï¿½neros**: `conj`($\alpha$), `itConj`($\alpha$).

### Operaciones bï¿½sicas de conjunto

Vacï¿½o() $\rightarrow$ $res$ : `conj`($\alpha$)
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \emptyset\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** genera un conjunto vacï¿½o.

Agregar(**in/out** $c$: `conj`($\alpha$), **in** $a : \alpha$) $\rightarrow$ $res$ : `itConj`($\alpha$)
**Pre** $\equiv$ $\{c =_{\text{obs}} c_0\}$
**Post** $\equiv$ $\{c =_{\text{obs}} Ag(a, c_0) \wedge \text{HaySiguiente}(res) \wedge_L \text{Siguiente}(res) = a \wedge \text{alias}(\text{esPermutacion?}(\text{SecuSuby}(res), c))\}$
**Complejidad:** $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
**Descripciï¿½n:** agrega el elemento $a$ al conjunto. Para poder acceder al elemento $a$ en $O(1)$, se devuelve un iterador a la posiciï¿½n de $a$ dentro de $c$.
**Aliasing:** el elemento $a$ se agrega por copia. El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n EliminarSiguiente. Ademï¿½s, anteriores($res$) y siguientes($res$) podrï¿½an cambiar completamente ante cualquier operaciï¿½n que modifique $c$ sin utilizar las funciones del iterador.

AgregarRapido(**in/out** $c$: `conj`($\alpha$), **in** $a : \alpha$) $\rightarrow$ $res$ : `itConj`($\alpha$)
**Pre** $\equiv$ $\{c =_{\text{obs}} c_0 \wedge a \notin c\}$
**Post** $\equiv$ $\{c =_{\text{obs}} Ag(a, c_0) \wedge \text{HaySiguiente}(res) \wedge_L \text{Siguiente}(res) = a \wedge \text{alias}(\text{esPermutacion?}(\text{SecuSuby}(res), c))\}$
**Complejidad:** $\Theta(copy(a))$

[4] Nï¿½tese que este es un abuso de notaciï¿½n, ya que no estamos describiendo *copy* y *equal* en funciï¿½n del tamaï¿½o de $a$. A la hora de usarlo, habrï¿½ que realizar la traducciï¿½n.

**Descripciï¿½n:** agrega el elemento $a \notin c$ al conjunto. Para poder acceder al elemento $a$ en $O(1)$, se devuelve un iterador a la posiciï¿½n de $a$ dentro de $c$.
**Aliasing:** el elemento $a$ se agrega por copia. El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n EliminarSiguiente. Ademï¿½s, anteriores($res$) y siguientes($res$) podrï¿½an cambiar completamente ante cualquier operaciï¿½n que modifique $c$ sin utilizar las funciones del iterador.

EsVacï¿½o?(**in** $c$: `conj(`$\alpha$`)`) $\to$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \emptyset?(c)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si $c$ esta vacï¿½o.

Pertenece?(**in** $c$: `conj(`$\alpha$`)`, **in** $a$: $\alpha$) $\to$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} a \in c)\}$
**Complejidad:** $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo $a$ pertenece al conjunto.

Eliminar(**in** $c$: `conj(`$\alpha$`)`, **in** $a$: $\alpha$)
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} c \setminus \{a\})\}$
**Complejidad:** $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
**Descripciï¿½n:** elimina $a$ de $c$, si es que estaba.

Cardinal(**in** $c$: `conj(`$\alpha$`)`) $\to$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \#c)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve la cantidad de elementos del conjunto.

Copiar(**in** $c$: `conj(`$\alpha$`)`) $\to$ $res$ : `conj(`$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} c\}$
**Complejidad:** $\Theta\left(\sum_{a \in c} copy(a)\right)$
**Descripciï¿½n:** genera una copia nueva del conjunto.

$\bullet = \bullet$(**in** $c_1$ : `conj(`$\alpha$`)`, **in** $c_2$: `conj(`$\alpha$`)`) $\to$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} c_1 = c_2\}$
**Complejidad:** $O\left(\sum_{a_1 \in c_1} \sum_{a_2 \in c_2} equal(a_1, a_2)\right)$.
**Descripciï¿½n:** compara $c_1$ y $c_2$ por igualdad.

## Operaciones del iterador

El iterador que presentamos permite modificar el conjunto recorrido, eliminando elementos. Sin embargo, cuando el conjunto es no modificable, no se pueden utilizar las funciones de eliminaciï¿½n. Ademï¿½s, los elementos iterados no pueden modificarse, por cuestiones de implementaciï¿½n.

CrearIt(**in** $c$: `conj(`$\alpha$`)`) $\to$ $res$ : `itConj(`$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {alias(esPermutacion?(SecuSuby($res$), $c$)) $\land$ vacia?(Anteriores($res$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** crea un iterador bidireccional del conjunto, de forma tal que HayAnterior evalï¿½e a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).

**Aliasing:** El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n EliminarSiguiente. Ademï¿½s, anteriores($res$) y siguientes($res$) podrï¿½an cambiar completamente ante cualquier operaciï¿½n que modifique $c$ sin utilizar las funciones del iterador.

HaySiguiente(**in** $it$: `itConj(`$\alpha$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ haySiguiente?($it$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si en el iterador todavï¿½a quedan elementos para avanzar.

HayAnterior(**in** $it$: `itConj(`$\alpha$`)`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ hayAnterior?($it$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si en el iterador todavï¿½a quedan elementos para retroceder.

Siguiente(**in** $it$: `itConj(`$\alpha$`)`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {HaySiguiente?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Siguiente($it$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento siguiente a la posiciï¿½n del iterador.
**Aliasing:** $res$ no es modificable.

Anterior(**in** $it$: `itConj(`$\alpha$`)`) $\rightarrow$ $res$ : $\alpha$
**Pre** $\equiv$ {HayAnterior?($it$)}
**Post** $\equiv$ {alias($res =_{\text{obs}}$ Anterior($it$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento anterior a la posiciï¿½n del iterador.
**Aliasing:** $res$ no es modificable.

Avanzar(**in/out** $it$: `itConj(`$\alpha$`)`)
**Pre** $\equiv$ {$it = it_0 \wedge$ HaySiguiente?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ Avanzar($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Avanza a la posiciï¿½n siguiente del iterador.

Retroceder(**in/out** $it$: `itConj(`$\alpha$`)`)
**Pre** $\equiv$ {$it = it_0 \wedge$ HayAnterior?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ Retroceder($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Retrocede a la posiciï¿½n anterior del iterador.

EliminarSiguiente(**in/out** $it$: `itConj(`$\alpha$`)`)
**Pre** $\equiv$ {$it = it_0 \wedge$ HaySiguiente?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ EliminarSiguiente($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Elimina de la lista iterada el valor que se encuentra en la posiciï¿½n siguiente del iterador.

EliminarAnterior(**in/out** $it$: `itConj(`$\alpha$`)`)
**Pre** $\equiv$ {$it = it_0 \wedge$ HayAnterior?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ EliminarAnterior($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Elimina de la lista iterada el valor que se encuentra en la posiciï¿½n anterior del iterador.

## Especificaciï¿½n de las operaciones auxiliares utilizadas en la interfaz

**TAD** Conjunto Extendido($\alpha$)

**extiende** Conjunto($\alpha$)

**otras operaciones (no exportadas)**
esPermutacion? : secu($\alpha$) $\times$ conj($\alpha$) $\longrightarrow$ bool

secuAConj : secu($\alpha$) $\longrightarrow$ conj($\alpha$)

**axiomas**

esPermutacion?($s, c$) $\equiv$ $c$ = secuAConj($s$) $\wedge$ $\#c$ = long($s$)

secuAConj($s$) $\equiv$ **if** vacia?($s$) **then** $\emptyset$ **else** Ag(prim($s$), secuAConj(fin($s$))) **fi**

**Fin TAD**

# Representacii¿½n

## Representacii¿½n del Conjunto

En este mï¿½dulo vamos a utilizar un diccionario lineal para representar el conjunto. La idea es que el conjunto de claves del diccionario represente el conjunto lineal. Si bien esta representacii¿½n no es la mï¿½s natural, permite resolver unas cuantas cuestiones sin duplicar codigo. La desventaja aparente es que gastamos memoria para guardar datos inï¿½tiles. Sin embargo, los lenguajes de programacii¿½n actuales permiten resolver este problema de forma mï¿½s o menos elegante. A nosotros no nos va a importar.

`conj`($\alpha$) **se representa con** `dicc(`$\alpha$`, bool)`

Rep : dicc($\alpha$,bool) $\longrightarrow$ bool

Rep($d$) $\equiv$ true

Abs : dicc($\alpha$,bool) $d$ $\longrightarrow$ conj($\alpha$) {Rep($d$)}

Abs($d$) $\equiv$ claves($d$)

## Representacii¿½n del iterador

El iterador del conjunto es simplemente un iterador del diccionario representante.

`itConj`($\alpha$) **se representa con** `itDicc(`$\alpha$`, bool)`

Rep : itDicc($\alpha$, bool) $\longrightarrow$ bool

Rep($it$) $\equiv$ true

Abs : itDicc($\alpha$, bool) $it$ $\longrightarrow$ itBi($\alpha$) {Rep($it$)}

Abs($it$) $=_{\text{obs}}$ b: itBi($\alpha$) | Anteriores($b$) = $\Pi_1$(Anteriores($it$)) $\wedge$ Siguientes($b$) = $\Pi_1$(Siguientes($it$))

$\Pi_1$ : secu(tupla($\alpha, \beta$)) $\longrightarrow$ secu($\alpha$)

$\Pi_1(s)$ $\equiv$ **if** vacia?($s$) **then** <> **else** $\Pi_1$(prim($s$)) $\bullet$ $\Pi_1$(Fin($s$)) **fi**

# Algoritmos

# 10. Mï¿½dulo Conjunto acotado de naturales

El mï¿½dulo conjunto acotado de naturales provee un conjunto en el que se pueden insertar ï¿½nicamente los elementos que se encuentran en un rango $[\ell, r]$ de naturales. La insercii¿½n, eliminacii¿½n y testeo de pertenencia de un elemento se pueden resolver en tiempo constante. El principal costo se paga cuando se crea la estructura, dado que cuesta tiempo lineal en $r - \ell$.

En cuanto al recorrido de los elementos, se provee un iterador bidireccional que tambii¿½n permite eliminar los elementos iterados.

# Especificacii¿½n

**TAD** Conjunto acotado

**géneros** conjAcotado

**igualdad observacional**

$$(\forall c_1, c_2 : \text{conjAcotado}) \left( c_1 =_{\text{obs}} c_2 \iff \begin{pmatrix} \text{Infimo}(c_1) =_{\text{obs}} \text{Infimo}(c_2) \wedge \text{Supremo}(c_1) =_{\text{obs}} \text{Supremo}(c_2) \wedge \\ \text{ConjSuby}(c_1) =_{\text{obs}} \text{ConjSuby}(c_2) \end{pmatrix} \right)$$

**observadores básicos**

Infimo : conjAcotado $\longrightarrow$ nat
Supremo : conjAcotado $\longrightarrow$ nat
ConjSuby : conjAcotado $\longrightarrow$ conj(nat)

**generadores**
$\emptyset$ : nat $\ell$ × nat $r$ $\longrightarrow$ conjAcotado $\{\ell \leq r\}$
Ag : nat $e$ × conjAcotado $c$ $\longrightarrow$ conjAcotado $\{\text{Infimo}(c) \leq e \leq \text{Supremo}(c)\}$

**otras operaciones**
Rango : conjAcotado $\longrightarrow$ tupla(nat, nat)

**axiomas**
Infimo($\emptyset(\ell, r)$) $\equiv$ $\ell$
Infimo(Ag($e, c$)) $\equiv$ Infimo($c$)
Supremo($\emptyset(\ell, r)$) $\equiv$ $r$
Supremo(Ag($e, c$)) $\equiv$ Supremo($c$)
ConjSuby($\emptyset(\ell, r)$) $\equiv$ $\emptyset$
ConjSuby(Ag($e, c$)) $\equiv$ Ag($e$, ConjSuby($c$))
Rango($c$) $\equiv$ $\langle$Infimo($c$), Supremo($c$)$\rangle$

**Fin TAD**

# Interfaz

**se explica con**: Conjunto acotado, Iterador Bidireccional(nat).

**gï¿½$\frac{1}{2}$neros**: `conjAcotado`, `itConjAcotado`.

## Operaciones bï¿½$\frac{1}{2}$sicas de conjunto

Vacï¿½$\frac{1}{2}$o(**in** $\ell$: `nat`, **in** $r$: `nat`) $\rightarrow$ $res$ : `conjAcotado`
**Pre** $\equiv$ $\{\ell \leq r\}$
**Post** $\equiv$ $\{res =_{\text{obs}} \emptyset(\ell, r)\}$
**Complejidad:** $\Theta(r - \ell)$
**Descripcï¿½$\frac{1}{2}$n:** genera un conjunto vacï¿½$\frac{1}{2}$o con el rango $[\ell, r.]$

Agregar(**in/out** $c$: `conjAcotado`, **in** $e$: `nat`)
**Pre** $\equiv$ $\{c =_{\text{obs}} c_0 \wedge \text{Infimo}(c) \leq e \leq \text{Supremo}(c)\}$
**Post** $\equiv$ $\{c =_{\text{obs}} Ag(e, c_0)\}$
**Complejidad:** $\Theta(1)$
**Descripcï¿½$\frac{1}{2}$n:** agrega el elemento $e$ al conjunto.

Infimo(**in** $c$: `conjAcotado`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \text{Infimo}(c)\}$
**Complejidad:** $\Theta(1)$
**Descripcï¿½$\frac{1}{2}$n:** devuelve el valor mï¿½$\frac{1}{2}$nimo que se puede agregar al conjunto.

Supremo(**in** $c$: `conjAcotado`) $\rightarrow$ $res$ : `nat`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \text{Supremo}(c)\}$
**Complejidad:** $\Theta(1)$
**Descripcï¿½$\frac{1}{2}$n:** devuelve el valor mï¿½$\frac{1}{2}$ximo que se puede agregar al conjunto.

EsVacï¿½$\frac{1}{2}$o?(**in** $c$: `conjAcotado`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} \emptyset?(\text{ConjSuby}(c))\}$
**Complejidad:** $\Theta(1)$
**Descripcï¿½$\frac{1}{2}$n:** devuelve `true` si y sï¿½$\frac{1}{2}$lo si $c$ esta vacï¿½$\frac{1}{2}$o.

Pertenece?(**in** $c$: `conjAcotado`, **in** $e$: `nat`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ $\{res =_{\text{obs}} e \in \text{ConjSuby}(c)\}$

**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo $e$ pertenece al conjunto. Notar que no es requerido que $e$ pertenezca al rango de $c$.

Eliminar(**in/out** $c$: `conjAcotado`, **in** $e$: `nat`)
**Pre** $\equiv \{c = c_0\}$
**Post** $\equiv \{\text{ConjSuby}(c) =_{\text{obs}} \text{ConjSuby}(c_0) \setminus \{e\} \land \text{Rango}(c) =_{\text{obs}} \text{Rango}(c_0)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Elimina $a$ de $c$, si es que estaba. Observar que no es requerido que $e$ pertenezca al rango de $c$.

Cardinal(**in** $c$: `conjAcotado`) $\to res$ : `nat`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \#\text{ConjSuby}(c)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Devuelve la cantidad de elementos del conjunto.

Copiar(**in** $c$: `conjAcotado`) $\to res$ : `conjAcotado`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} c\}$
**Complejidad:** $\Theta(\text{Supremo}(c)$ - $\text{Infimo}(c))$
**Descripciï¿½n:** genera una copia nueva del conjunto.

$\bullet = \bullet$(**in** $c_1$ : `conjAcotado`, **in** $c_2$ : `conjAcotado`) $\to res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} c_1 = c_2\}$
**Complejidad:** $\Theta(\text{mín}\{\#c_1, \#c_2\})$.
**Descripciï¿½n:** compara $c_1$ y $c_2$ por igualdad.

## Operaciones del iterador

El iterador que presentamos permite modificar el conjunto recorrido, eliminando elementos. Sin embargo, cuando el conjunto es no modificable, no se pueden utilizar las funciones de eliminaciï¿½n. Todos los naturales del conjunto son iterados por copia.

CrearIt(**in** $c$: `conjAcotado`) $\to res$ : `itConjAcotado`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv$ {alias(esPermutaciï¿½n?(SecuSuby($res$), ConjSuby($c$))) $\land$ vacia?(Anteriores($res$))}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** crea un iterador bidireccional del conjunto, de forma tal que HayAnterior evalï¿½e a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).
**Aliasing:** El iterador se invalida si y sï¿½lo si se elimina el elemento siguiente del iterador sin utilizar la funciï¿½n EliminarSiguiente. Ademï¿½s, anteriores($res$) y siguientes($res$) podrï¿½an cambiar completamente ante cualquier operaciï¿½n que modifique $c$ sin utilizar las funciones del iterador.

HaySiguiente(**in** $it$: `itConjAcotado`) $\to res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{haySiguiente?}(it)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si en el iterador todavï¿½a quedan elementos para avanzar.

HayAnterior(**in** $it$: `itConjAcotado`) $\to res$ : `bool`
**Pre** $\equiv \{\text{true}\}$
**Post** $\equiv \{res =_{\text{obs}} \text{hayAnterior?}(it)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve `true` si y sï¿½lo si en el iterador todavï¿½a quedan elementos para retroceder.

Siguiente(**in** $it$: `itConjAcotado`) $\to res$ : `nat`
**Pre** $\equiv \{\text{HaySiguiente?}(it)\}$
**Post** $\equiv \{res =_{\text{obs}} \text{Siguiente}(it)\}$
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento siguiente a la posiciï¿½n del iterador.

**Aliasing:** *res* se devuelve por copia.

ANTERIOR(**in** *it*: `itConjAcotado`) $\rightarrow$ *res* : `nat`
**Pre** $\equiv$ {HayAnterior?($it$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ Anterior($it$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** devuelve el elemento anterior a la posiciï¿½n del iterador.
**Aliasing:** *res* se devuelve por copia.

AVANZAR(**in/out** *it*: `itConjAcotado`)
**Pre** $\equiv$ {$it = it_0 \wedge$ HaySiguiente?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ Avanzar($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Avanza a la posiciï¿½n siguiente del iterador.

RETROCEDER(**in/out** *it*: `itConjAcotado`)
**Pre** $\equiv$ {$it = it_0 \wedge$ HayAnterior?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ Retroceder($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Retrocede a la posiciï¿½n anterior del iterador.

ELIMINARSIGUIENTE(**in/out** *it*: `itConjAcotado`)
**Pre** $\equiv$ {$it = it_0 \wedge$ HaySiguiente?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ EliminarSiguiente($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Elimina del conjunto el elemento que se encuentra en la posiciï¿½n siguiente.

ELIMINARANTERIOR(**in/out** *it*: `itConjAcotado`)
**Pre** $\equiv$ {$it = it_0 \wedge$ HayAnterior?($it$)}
**Post** $\equiv$ {$it =_{\text{obs}}$ EliminarAnterior($it_0$)}
**Complejidad:** $\Theta(1)$
**Descripciï¿½n:** Elimina del conjunto el elemento que se encuentra en la posiciï¿½n anterior.

# Representaciï¿½n

## Representaciï¿½n del Conjunto

La idea de este mï¿½dulo es aprovechar que los elementos que se pueden llegar a agregar son naturales en un rango que se conoce desde el inicio, de forma tal de poder acceder a ellos en tiempo $O(1)$. Para esto, podemos tener un arreglo $a$ de booleanos de tamaï¿½o $r - \ell + 1$ de forma tal que $\ell \leq e \leq r$ pertenezca al conjunto si y sï¿½lo si $a[e - \ell]$ = true. El inconveniente de esta representaciï¿½n es que no permite iterar todos los elementos en tiempo lineal en la cantidad de elementos del conjunto. En efecto, si el conjunto tiene un ï¿½nico elemento $e$, igual tenemos que recorrer todo el rango $r - \ell$ (que no es constante) para encontrar $e$. Para subsanar este inconveniente, vamos a guardar un conjunto lineal $c$ con los elementos que pertenecen al conjunto acotado. Para poder eliminar el elemento $e$, debemos poner en false el valor de $a[e - \ell]$, a la vez que tenemos que eliminar a $c$ del conjunto. Esto se puede hacer en tiempo $O(1)$ si podemos obtener eficientemente un "puntero" a $e$ dentro de $c$. Este puntero podrï¿½a ser un iterador. Luego, en $a$ vamos a tener, ademas del booleano, un iterador al conjunto $c$ que nos permita acceder en $O(1)$ a $e$ dentro de $c$. Una mejora a esta estructura es eliminar el booleano de $a$, y considerar que $e$ pertenece al conjunto acotado si y sï¿½lo si el iterador de $a[e - \ell]$ tiene un elemento siguiente. Este elemento siguiente contiene a $e$ en $c$.

`conjAcotado` **se representa con** `ca`

donde `ca` es `tupla`(*pertenencia*: `arreglo_dimensionable de iterConj(nat)`,
*elementos*: `conj(nat)`, *infimo*: `nat`)

Rep : ca $\longrightarrow$ bool
Rep($c$) $\equiv$ true $\iff$ ($\forall e$: nat)($e \in c$.elementos $\iff e \geq c$.infimo $\wedge$ $e < c$.infimo + tam($c$.pertenencia) $\wedge_{\text{L}}$
HaySiguiente?($c$.pertenencia[$e - c$.infimo])) $\wedge_{\text{L}}$
($\forall e$: nat)($e \in c$.elementos $\Rightarrow_{\text{L}}$ Siguiente($c$.pertenencia[$e - c$.infimo]) = $e$)

Abs : ca $e$ $\longrightarrow$ conjAcotado {Rep($e$)}

Abs($e$) $=_{\text{obs}}$ c: conjAcotado | Infimo($c$) = $e$.infimo ∧ Supremo($c$) = $e$.infimo + tam($e$.pertenencia) − 1 ∧
ConjSuby($c$) = $e$.elementos

## Representaciï¿½n del iterador

El iterador del conjunto acotado es simplemente un iterador del conjunto *elementos*, ya que con ï¿½ste recorremos todos los elementos, mï¿½s un puntero a la estructura del conjunto, para poder borrar al eliminar el iterador.

**`itConjAcotado` se representa con `itCA`**

donde `itCA` es `tupla`(*iter*: `itConj(nat)`, *conj*: `puntero(ca)`)

Rep : itCA ⟶ bool
Rep($it$) ≡ true ⟺ Rep(∗$it$.conj) ∧ EsPermutacion(SecuSuby($it$.iter), $it$.conj→elementos)

Abs : itCA $it$ ⟶ itBi(nat) {Rep($it$)}
Abs($it$) ≡ $it$.elementos

# Algoritmos