Apunte de Módulos Básicos (v. 0.3α)

Algoritmos y Estructuras de Datos II, DC, UBA.

1er cuatrimestre de 2019

Índice

1. Introducción **2**

2. TADs para especificar iteradores **2**
- 2.1. TAD Iterador Unidireccional(α) 2
- 2.2. TAD Iterador Unidireccional Modificable(α) 2
- 2.3. Iterador Bidireccional(α) 3

3. Invariantes de aliasing **4**

4. Módulo Lista Enlazada(α) **5**

5. Módulo Pila(α) **11**

6. Módulo Cola(α) **12**

7. Módulo Vector(α) **14**

8. Módulo Diccionario Lineal(κ, σ) **17**

9. Módulo Conjunto Lineal(α) **22**

10. Módulo Conjunto acotado de naturales **25**

1. Introducción

El presente documento describe varios módulos que se pueden utilizar para realizar el TP de diseño. Además, sirve como ejemplo de qué se espera del TP de diseño, y muestra algunas técnicas que podrían ser útiles a la hora de desarrollar nuevos módulos.

Antes de introducir los módulos, se especifican los tipos de iteradores que se van a utilizar. Esta especificación es auxiliar, para simplificar las precondiciones y postcondiciones de los algoritmos que utilizan iteradores. Luego, se presentan todos los módulos, con su interfaz, representación y cálculos de complejidad.

NOTA: Este apunte no está terminado. Además de ser incompleto (faltan los algoritmos y los cálculos de complejidad de todos los módulos), puede tener (mejor dicho, tiene) errores y podría sufrir cambios en cualquier momento.

2. TADs para especificar iteradores

En esta sección se describen los TADs que utilizamos en la materia para especificar los iteradores. Los mismos no son más 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 especificación de los iteradores con TADs podría evitarse, pero lo incluimos para simplificar la especificación de los módulos.

2.1. TAD Iterador Unidireccional(α)

El iterador unidireccional permite recorrer los elementos una única vez, avanzando continuamente. Es el tipo de iterador más 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 especificación. La idea final es que esta secuencia describa el orden en el que se recorrerán los elementos de la estructura, i.e., esta secuencia es una "permutación" de la estructura iterada.

TAD Iterador Unidireccional(α)

parámetros formales
géneros α

géneros itUni(α)

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(α) $\longrightarrow$ secu(α)

generadores
CrearItUni : secu(α) $\longrightarrow$ itUni(α)

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

El iterador unidireccional modificable es una extensión del iterador unidireccional que permite realizar algunas operaciones de modificación 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 cómo es el aliasing entre el iterador y el tipo iterado en el módulo correspondiente.

TAD Iterador Unidireccional Modificable(α)

parámetros formales

géneros α

géneros itMod(α)

igualdad observacional

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

observadores básicos

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

generadores

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

otras operaciones

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

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(α)

El iterador bidireccional es una generalización del iterador unidireccional modificable. El mismo permite recorrer los elementos avanzando y retrocediendo. Si bien se podría hacer una versión de iterador bidireccional no modificable, la especificación de ambas es similar. Cuando se utilice en un módulo que no permita algunas modificaciones, simplemente se puede omitir el diseño de las funciones que realizan estas modificaciones (ver e.g., módulo Conjunto Lineal). Por este motivo, optamos sólo por la versión modificable.

Para que el iterador bidireccional sea lo mas simétrico posible, cambiamos la operación actual por dos: anterior y siguiente. La idea conceptual es pensar que el iterador está posicionado en el medio de dos posiciones, y puede acceder tanto a la anterior como a la siguiente. Obviamente, la implementación puede diferir de esta visión conceptual.

TAD Iterador Bidireccional(α)

parámetros formales

géneros α

géneros itBi(α)

igualdad observacional

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

observadores básicos

Anteriores : itBi(α) $\longrightarrow$ secu(α)
Siguientes : itBi(α) $\longrightarrow$ secu(α)

generadores

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

otras operaciones

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

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 descripción 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 ejecución del programa. Si bien el metapredicado alias no es parte del lenguaje de TADs y no lo describimos en lógica de primer orden, lo vamos a utilizar en las precondiciones y postcondiciones de las funciones. En esta sección vamos a describir su semántica en castellano.

Alias es un metapredicado con un único parámetro ϕ que puede ser una expresión booleana del lenguaje de TADs o un predicado en lógica de primer orden. Este paramétro ϕ involucrará un conjunto V con dos o más variables del programa. El significado es que las variables de V satisfacen ϕ durante la ejecución 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 está siendo iterado.

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

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

lo que significa informalmente es que s y t comparten la misma posición de memoria. Un poco más rigurosamente, lo que significa es que cualquier modificación 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 básico. Supongamos ahora que tenemos dos variables s y c de tipos secu(α) y conj(α), respectivamente. Si escribimos

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

estamos diciendo que s y c comparten la misma memoria de forma tal que cualquier modificación 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 permutación de $c \cup \{a\}$. Notemos que, en particular, s

podría cambiar a cualquier permutación, salvo que se indique lo contrario. De la misma forma, si se eliminara un elemento a de s, entonces c tambien se vería afectado de forma tal que s sea una permutación de c. En particular, c pasaría a ser $c \setminus \{a\}$.

Debemos observar que este invariante no es magico, sino que es una declaración como cualquier otra, y el programado debe asegurarse que este invariante se cumpla. En particular, en el ejemplo anterior, no deberiamos permitir la inserción de elementos repetido en s, ya que dejaría de ser una posible permutación de un conjunto.

4. Módulo Lista Enlazada(α)

Interfaz

generos: `mapa`.

se explica con: Secuencia(α), Iterador Bidireccional(α).

géneros: `lista`(α), `itLista`(α).

Operaciones básicas de lista

Representación

Representación de la lista

El objetivo de este módulo 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 último apunta al primero y el anterior del primero apunta al último. La estructura de representación, su invariante de representación y su función de abstracción son las siguientes.

`mapa` se representa con `m`

donde `m` es `tupla`(*tamano*: `nat`, *casilleros*: `vec(vec(bool))`)

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

$\text{Abs} : \text{mapa } m \longrightarrow \text{hab}$ $\qquad \{\text{Rep}(m)\}$
$\text{Abs}(m) \equiv m.tamano =_{\text{obs}} tam(h) \land_{\text{L}} (\forall t\text{: tuple(nat,nat)})(0 \leq \Pi_1(t), \Pi_2(t) < m.tamano - 1 \Rightarrow_{\text{L}} \text{libre(m, t)} =_{\text{obs}}$
$\quad m.casilleros[\Pi_1(t)][\Pi_2(t)])$

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

`itLista`(α) se representa con `iter`

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

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

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

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

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

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

Pos : lst l × puntero(nodo) p ⟶ puntero(nodo) {Rep(⟨l, p⟩)}

Pos(l,p) ≡ **if** l.primero = p ∨ l.longitud = 0 **then** 0 **else** 1 + Pos(FinLst(l), p) **fi**

Algoritmos

En esta sección se hace abuso de notación en los cálculos de álgebra de órdenes presentes en la justificaciones de los algoritmos. La operación de suma "+" denota secuencialización de operaciones con determinado orden de complejidad, y el símbolo de igualdad "=" denota la pertenencia al orden de complejidad resultante.

Algoritmos del módulo

---

iVacía() → *res* : lst

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

Complejidad: $\Theta(1)$

---

iAgregarAdelante(**in/out** l : `lst`, **in** a : α) → *res* : *iter*

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

Complejidad: $\Theta(copy(a))$
Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(copy(a))$. Aplicando álgebra de órdenes: $\Theta(1) + \Theta(1) + \Theta(copy(a)) = \Theta(copy(a))$

---

iAgregarAtras(**in/out** l : `lst`, **in** a : α) → *res* : *iter*

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

Complejidad: $\Theta(copy(a))$
Justificación: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(copy(a))$. Aplicando álgebra de órdenes: $\Theta(1) + \Theta(copy(a)) + \Theta(1) = \Theta(copy(a))$

---

iEsVacía?(**in** l : `lst`) → *res* : *bool*

$res \leftarrow (l.primero = NULL)$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

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

$CrearIt(l).EliminarSiguiente()$ ▷ $\Theta(1) + \Theta(1)$

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

---

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

$CrearItUlt(l).EliminarAnterior()$ ▷ $\Theta(1) + \Theta(1)$

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

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

$res \leftarrow CrearIt(l).Siguiente()$ ▷ $\Theta(1) + \Theta(1)$

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

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

$res \leftarrow CrearItUlt(l).Anterior()$ ▷ $\Theta(1) + \Theta(1)$

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

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

$res \leftarrow l.longitud$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

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

$it \leftarrow CrearIt(l)$ ▷ $\Theta(1)$
$indice \leftarrow 0$ ▷ $\Theta(1)$
while $indice < i$ **do** ▷ $\Theta(i)$
    $Avanzar(it)$ ▷ $\Theta(1)$
    $indice \leftarrow indice + 1$ ▷ $\Theta(1)$
end while
$res \leftarrow Siguiente(it)$ ▷ $\Theta(1)$

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

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

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

Complejidad: $\Theta\left(\sum_{i=1}^{long(l)} copy(l[i])\right)$
Justificación: El algoritmo cuenta con un ciclo que se repetirá long(l) veces (recorre la lista entera). Por cada ciclo realiza una copia del elemento, el costo será el de copiar el elemento. Por lo tanto, el costo total del ciclo será 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`) $\to$ *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 $\ell = \text{mín}\{\text{long}(l_1), \text{long}(l_2)\}$. [$*$]

Justificación: [$*$] 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`) $\to$ *res* : iter
$res \leftarrow \langle l.primero, l\rangle$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

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

Complejidad: $\Theta(1)$

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

Complejidad: $\Theta(1)$

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

Complejidad: $\Theta(1)$

iSiguiente(**in** it : `iter`) $\to$ *res* : α
$res \leftarrow (it.siguiente \to dato)$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

iAnterior(**in** it : `iter`) $\to$ *res* : α
$res \leftarrow (SiguienteReal(it) \to anterior \to dato)$ $\triangleright\ \Theta(1)$

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: $\Theta(1)$
Justificación: $\Theta(1) + \Theta(1) = \Theta(1)$

---

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 último 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: $\Theta(1)$
Justificación: $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

---

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

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

---

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

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

iAgregarComoAnterior(**in/out** it : `iter`, **in** a : α)
$puntero(nodo)\ sig \leftarrow SiguienteReal(it)$
$puntero(nodo)\ nuevo \leftarrow$ **&** $\langle a, NULL, NULL \rangle$ ▷ 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 // $\Theta(1)$
  $(nuevo \to anterior) \leftarrow nuevo$
  $(nuevo \to siguiente) \leftarrow nuevo$
else
  $(nuevo \to anterior) \leftarrow (sig \to anterior)$
  $(nuevo \to siguiente) \leftarrow sig$
end if
$(nuevo \to anterior \to siguiente) \leftarrow nuevo$ ▷ Reencadenamos los otros nodos // $\Theta(1)$
if $it.siguiente = (it.lista \to primero)$ **then** ▷ Cambiamos el primero en caso de que estemos agregando el primero // $\Theta(1)$
  $(it.lista \to primero) \leftarrow nuevo$
end if
$(it.lista \to longitud) \leftarrow (it.lista \to longitud) + 1$ ▷ $\Theta(1)$

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

---

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

Complejidad: $\Theta(1)$

5. Módulo Pila(α)

El módulo Pila provee una pila en la que sólo 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 función de α en $\mathbb{N}$).[1]

Interfaz

parámetros formales
géneros α
función COPIAR(**in** $a : \alpha$) $\rightarrow$ res : α
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} a\}$
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's

se explica con: PILA(α).

géneros: `pila`(α).

VACÍA() $\rightarrow$ res : `pila`(α)
Pre $\equiv$ {}
Post $\equiv$ $\{res =_{\text{obs}} \text{vacía}\}$
Complejidad: $\Theta(1)$
Descripción: genera una pila vacía.

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

ESVACIA?(**in** p : `pila`(α)) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{vacia?}(p)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si la pila no contiene elementos

TOPE(**in** p : `pila`(α)) $\rightarrow$ res : α
Pre $\equiv$ $\{\neg\text{vacía?}(p)\}$
Post $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{tope}(p))\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el tope de la pila.
Aliasing: res es modificable si y sólo si p es modificable.

DESAPILAR(**in/out** p : `pila`(α)) $\rightarrow$ res : α
Pre $\equiv$ $\{p =_{\text{obs}} p_0 \wedge \neg\text{vacía?}(p)\}$
Post $\equiv$ $\{p =_{\text{obs}} \text{desapilar}(p_0) \wedge res =_{\text{obs}} \text{tope}(p)\}$
Complejidad: $\Theta(1)$
Descripción: desapila el tope de p.

TAMAÑO(**in** p : `pila`(α)) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{tamaño}(p)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de elementos apilados en p.

[1] Nótese que este es un abuso de notación, ya que no estamos describiendo *copy* en función del tamaño de a. A la hora de usarlo, habrá que realizar la traducción

COPIAR(**in** p: $\texttt{pila}(\alpha)$) $\rightarrow$ res : $\texttt{pila}(\alpha)$
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 = tamaño(p).
Descripción: genera una copia nueva de la pila

$\bullet = \bullet$(**in** p_1 : $\texttt{pila}(\alpha)$, **in** p_2 : $\texttt{pila}(\alpha)$) $\rightarrow$ res : $\texttt{bool}$
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 $t = \text{mín}\{\text{tamaño}(p_1), \text{tamaño}(p_2)\}$.
Descripción: compara p_1 y p_2 por igualdad, cuando α posee operación de igualdad.
Requiere: $\bullet = \bullet$(**in** a_1 : α, **in** a_2 : α) $\rightarrow$ res : $\texttt{bool}$
  Pre $\equiv$ {true}
  Post $\equiv$ $\{res =_{\text{obs}} (a_1 = a_2)\}$
  Complejidad: $\Theta(equal(a_1, a_2))$
  Descripción: función de igualdad de α's

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Pila Extendida(α)

extiende PILA(α)

otras operaciones (no exportadas)

$\bullet[\bullet]$: pila(α) p $\times$ nat i $\longrightarrow$ α $\{i < \text{tamaño}(p)\}$

axiomas

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

Fin TAD

Representación

El objetivo de este módulo es implementar una pila lo más 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 último elemento. En este caso, elegimos que el tope se encuentre en el primer elemento.

$\texttt{pila}(\alpha)$ **se representa con** $\texttt{lista}(\alpha)$

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

Abs : lista(α) l $\longrightarrow$ pila(α) {Rep(l)}
Abs(l) $\equiv$ **if** vacia?(l) **then** vacía **else** apilar(prim(l), Abs(fin(l))) **fi**

Algoritmos

6. Módulo Cola(α)

El módulo Cola provee una cola en la que sólo 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 función de α en $\mathbb{N}$).[2]

Interfaz

[2] Nótese que este es un abuso de notación, ya que no estamos describiendo *copy* en función del tamaño de *a*. A la hora de usarlo, habrá que realizar la traducción

parámetros formales
géneros α
función COPIAR(**in** $a : \alpha$) $\to res : \alpha$
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} a\}$
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's

se explica con: COLA(α).

géneros: `cola`(α).

VACÍA() $\to res$: `cola`(α)
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}}$ vacía$\}$
Complejidad: $\Theta(1)$
Descripción: genera una cola vacía.

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

ESVACIA?(**in** c: `cola`(α)) $\to res$: `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{vacia?}(c)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si la cola es vacía.

PROXIMO(**in** c: `cola`(α)) $\to res : \alpha$
Pre $\equiv$ $\{\not\text{vacía?}(c)\}$
Post $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{proximo}(c))\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el proximo de la cola.
Aliasing: res es modificable si y sólo si p es modificable.

DESENCOLAR(**in/out** c: `cola`(α))
Pre $\equiv$ $\{c =_{\text{obs}} c_0 \wedge \neg\text{vacía?}(c)\}$
Post $\equiv$ $\{c =_{\text{obs}} \text{desacolar}(c_0)\}$
Complejidad: $\Theta(1)$
Descripción: desencola el proximo de c.

TAMAÑO(**in** c: `cola`(α)) $\to res$: `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{tamaño}(c)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de elementos encolados en c.

COPIAR(**in** c: `cola`(α)) $\to res$: `cola`(α)
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} c\}$
Complejidad: $\Theta\left(\sum_{i=1}^{t} copy(c[i])\right)$, donde $t = \text{tamaño}(c)$
Descripción: genera una copia nueva de la cola

$\bullet = \bullet$(**in** c_1 : `cola`(α), **in** c_2 : `cola`(α)) $\to res$: `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 $t = \min\{\text{tamaño}(c_1), \text{tamaño}(c_2)\}$.
Descripción: compara c_1 y c_2 por igualdad, cuando α posee operación 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: $\Theta(equal(a_1, a_2))$
Descripción: función de igualdad de α's

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Cola Extendida(α)

extiende COLA(α)

otras operaciones (no exportadas)

$\bullet[\bullet]$: cola(α) p × nat i ⟶ α $\{i < \text{tamaño}(p)\}$

axiomas

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

Fin TAD

Representación

El objetivo de este módulo es implementar una cola lo más 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 último elemento. En este caso, elegimos que el proximo se encuentre en el primer elemento.

`cola`(α) **se representa con** `lista`(α)

Rep : lista(α) ⟶ bool
Rep(l) ≡ true

Abs : lista(α) l ⟶ cola(α) {Rep(l)}
Abs(l) ≡ **if** vacia?(l) **then** vacía **else** encolar(ult(l), Abs(com(l))) **fi**

Algoritmos

7. Módulo Vector(α)

El módulo Vector provee una secuencia que permite obtener el i-ésimo elemento de forma eficiente. La inserción de elementos es eficiente cuando se realiza al final de la misma, si se utiliza un análisis amortizado (i.e., n inserciones consecutivas cuestan $O(n)$), aunque puede tener un costo lineal en peor caso. La inserción en otras posiciones no es tan eficiente, ya que requiere varias copias de elementos. El borrado de los últimos elementos es eficiente, no así el borrado de los elementos intermedios.

Una consideración a tener en cuenta, es que el espacio utilizado por la estructura es el máximo 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 α. 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 índice, 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 función de α en $\mathbb{N}$), y vamos a utilizar

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

para describir el costo de inserción 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 inserción consecutiva de n elementos costará $O(1)$ copias por elemento, en términos asintóticos.

Interfaz

parámetros formales

géneros α

función $\text{Copiar}(\textbf{in } a : \alpha) \to res : \alpha$

Pre $\equiv$ {true}

Post $\equiv \{res =_{\text{obs}} a\}$

Complejidad: $\Theta(copy(a))$

Descripción: función de copia de α's.

se explica con: $\text{Secu}(\alpha)$.

géneros: `vector`(α).

$\text{Vacía}() \to res : \texttt{vector}(\alpha)$

Pre $\equiv$ {true}

Post $\equiv \{res =_{\text{obs}} <>\}$

Complejidad: $\Theta(1)$

Descripción: genera un vector vacío.

$\text{AgregarAtras}(\textbf{in/out } v : \texttt{vector}(\alpha), \textbf{in } a : \alpha)$

Pre $\equiv \{v =_{\text{obs}} v_0\}$

Post $\equiv \{v =_{\text{obs}} v_0 \circ a\}$

Complejidad: $\Theta(f(\text{long}(v)) + copy(a))$

Descripción: agrega el elemento a como último 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.

$\text{EsVacío?}(\textbf{in } v : \texttt{vector}(\alpha)) \to res : \texttt{bool}$

Pre $\equiv$ {true}

Post $\equiv \{res =_{\text{obs}} \text{vacia?}(v)\}$

Complejidad: $\Theta(1)$

Descripción: devuelve `true` si y sólo si v esta vacío

$\text{Comienzo}(\textbf{in/out } v : \texttt{vector}(\alpha))$

Pre $\equiv \{v =_{\text{obs}} v_0 \land \neg\text{vacía?}(v)\}$

Post $\equiv \{v =_{\text{obs}} \text{com}(v_0)\}$

Complejidad: $\Theta(1)$

Descripción: elimina el último elemento de v.

$\text{TomarPrimeros}(\textbf{in/out } v : \texttt{vector}(\alpha), \textbf{in } n : \texttt{nat})$

Pre $\equiv \{v =_{\text{obs}} v_0\}$

Post $\equiv \{v =_{\text{obs}} \text{Tomar}(v_0, n)\}$

Complejidad: $\Theta(1)$

Descripción: elimina los últimos máx{long(v) - n, 0} elementos del vector, i.e., se queda con los primeros n elementos del vector.

$\text{TirarUltimos}(\textbf{in/out } v : \texttt{vector}(\alpha), \textbf{in } n : \texttt{nat})$

Pre $\equiv \{v =_{\text{obs}} v_0\}$

Post $\equiv \{v =_{\text{obs}} \text{Tomar}(v_0, \text{long}(v_0) - n)\}$

Complejidad: $\Theta(1)$

Descripción: elimina los últimos máx{long(v), n} elementos del vector.

$\text{Ultimo}(\textbf{in } v : \texttt{vector}(\alpha)) \to res : \alpha$

Pre $\equiv \{\neg\text{vacía?}(v)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{ult}(v))\}$

Complejidad: $\Theta(1)$

Descripción: devuelve el último elemento del vector.

Aliasing: res es modificable si y sólo si v es modificable.

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

$\bullet[\bullet]$(**in** v : `vector(`α`)`, **in** i : `nat`) $\rightarrow$ res : α
Pre $\equiv$ $\{i < \text{long}(v)\}$
Post $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{iesimo}(v, i))\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento que se encuentra en la i-ésima posición del vector en base 0. Es decir, $v[i]$ devuelve el elemento que se encuentra en la posición $i + 1$.
Aliasing: res es modificable si y sólo si v es modificable.

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

ELIMINAR(**in/out** v : `vector(`α`)`, **in** i : `nat`)
Pre $\equiv$ $\{v =_{\text{obs}} v_0 \wedge i < \text{long}(v)\}$
Post $\equiv$ $\{v =_{\text{obs}} \text{Eliminar}(v, i)\}$
Complejidad: $\Theta(\text{long}(v) - i)$
Descripción: elimina el elemento que ocupa la posición i de v.

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

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

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Secuencia Extendida(α)

extiende SECUENCIA(α)

otras operaciones (exportadas)

Agregar	: secu(α) s × nat i × α a	$\longrightarrow$	secu(α)	$\{i \leq \text{long}(s)\}$
Eliminar	: secu(α) s × nat i	$\longrightarrow$	secu(α)	$\{i < \text{long}(s)\}$
Tomar	: secu(α) × nat	$\longrightarrow$	secu(α)	

otras operaciones (no exportadas)

Tirar : secu(α) × nat ⟶ secu(α)

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

Fin TAD

Representación

La idea de este módulo es tener una lista donde el i-ésimo se puede obtener en tiempo $O(1)$. Para esto, necesitamos usar algún 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 tamaño del arreglo cuando este se llena.

`vector`(α) se representa con `vec`

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

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

Abs : vec v ⟶ secu(α) {Rep(v)}
Abs(v) ≡ **if** v.longitud = 0 **then**
<>
else
Abs(⟨v.elementos, v.longitud − 1⟩) ∘ ∗(v.elementos[v.longitud − 1])
fi

Algoritmos

8. Módulo Diccionario Lineal(κ, σ)

El módulo Diccionario Lineal provee un diccionario básico en el que se puede definir, borrar, y testear si una clave está definida en tiempo lineal. Cuando ya se sabe que la clave a definir no esta definida en el diccionario, la definición 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 κ, σ.

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

parámetros formales

géneros κ, σ

función • = •(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) → *res* : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} (k_1 = k_2)$}
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ's

función COPIAR(**in** $k : \kappa$) → *res* : κ
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} k$}
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

[3] Nótese que este es un abuso de notación, ya que no estamos describiendo *copy* y *equal* en función del tamaño de k. A la hora de usarlo, habrá que realizar la traducción.

función $\textsc{Copiar}(\textbf{in } s\colon \sigma) \to res : \sigma$
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} s\}$
Complejidad: $\Theta(copy(s))$
Descripción: función de copia de σ's

se explica con: $\textsc{Diccionario}(\kappa, \sigma)$, $\textsc{Iterador Bidireccional}(\textsc{tupla}(\kappa, \sigma))$.

géneros: `dicc`(κ, σ), `itDicc`(κ, σ).

Operaciones básicas de diccionario

$\textsc{Vacío}() \to res$: `dicc`(κ, σ)
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{vacio}\}$
Complejidad: $\Theta(1)$
Descripción: genera un diccionario vacío.

$\textsc{Definir}(\textbf{in/out } d\colon$ `dicc`(κ, σ), $\textbf{in } k\colon \kappa$, $\textbf{in } s\colon \sigma) \to res$: `itDicc`(κ, σ)
Pre $\equiv \{d =_{\text{obs}} d_0\}$
Post $\equiv \{d =_{\text{obs}} \text{definir}(d, k, s) \wedge \text{haySiguiente}(res) \wedge_L \text{Siguiente}(res) = \langle k, s\rangle \wedge \text{alias}(\text{esPermutación}(\text{SecuSuby}(res), d))\}$
Complejidad: $\Theta\left(\sum_{k' \in K} equal(k, k') + copy(k) + copy(s)\right)$, donde $K = \text{claves}(d)$
Descripción: define la clave k con el significado s en el diccionario. Retorna un iterador al elemento recién agregado.
Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función $\textsc{EliminarSiguiente}$. Además, anteriores($res$) y siguientes($res$) podrían cambiar completamente ante cualquier operación que modifique el d sin utilizar las funciones del iterador.

$\textsc{DefinirRapido}(\textbf{in/out } d\colon$ `dicc`(κ, σ), $\textbf{in } k\colon \kappa$, $\textbf{in } s\colon \sigma) \to res$: `itDicc`(κ, σ)
Pre $\equiv \{d =_{\text{obs}} d_0 \wedge \neg\text{definido?}(d, k)\}$
Post $\equiv \{d =_{\text{obs}} \text{definir}(d, k, s) \wedge \text{haySiguiente}(res) \wedge_L \text{Siguiente}(res) = \langle k, s\rangle \wedge \text{esPermutación}(\text{SecuSuby}(res), d)\}$
Complejidad: $\Theta(copy(k) + copy(s))$
Descripción: define la clave $k \notin \text{claves}(d)$ con el significado s en el diccionario. Retorna un iterador al elemento recién agregado.
Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función $\textsc{EliminarSiguiente}$. Además, anteriores($res$) y siguientes($res$) podrían cambiar completamente ante cualquier operación que modifique el d sin utilizar las funciones del iterador.

$\textsc{Definido?}(\textbf{in } d\colon$ `dicc`(κ, σ), $\textbf{in } k\colon \kappa) \to res$: `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{def?}(d, k)\}$
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K = \text{claves}(d)$
Descripción: devuelve `true` si y sólo k está definido en el diccionario.

$\textsc{Significado}(\textbf{in } d\colon$ `dicc`(κ, σ), $\textbf{in } k\colon \kappa) \to res : \sigma$
Pre $\equiv \{\text{def?}(d, k)\}$
Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Significado}(d, k))\}$
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K = \text{claves}(d)$
Descripción: devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.

$\textsc{Borrar}(\textbf{in/out } d\colon$ `dicc`(κ, σ), $\textbf{in } k\colon \kappa)$
Pre $\equiv \{d = d_0 \wedge \text{def?}(d, k)\}$
Post $\equiv \{d =_{\text{obs}} \text{borrar}(d_0, k)\}$
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K = \text{claves}(d)$
Descripción: elimina la clave k y su significado de d.

$\#\textsc{Claves}(\textbf{in } d\colon \texttt{dicc}(\kappa, \sigma)) \to res : \texttt{nat}$
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \#\text{claves}(d)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de claves del diccionario.

$\textsc{Copiar}(\textbf{in } d\colon \texttt{dicc}(\kappa, \sigma)) \to res : \texttt{dicc}(\kappa, \sigma)$
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 $K = \text{claves}(d)$
Descripción: genera una copia nueva del diccionario.

$\bullet = \bullet(\textbf{in } d_1 : \texttt{dicc}(\kappa, \sigma), \textbf{in } d_2 : \texttt{dicc}(\kappa, \sigma)) \to res : \texttt{bool}$
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 $K_i = \text{claves}(d_i)$ y $s_i = \text{significado}(d_i, k_i)$, $i \in \{1, 2\}$.
Descripción: compara d_1 y d_2 por igualdad, cuando σ posee operación de igualdad.
Requiere: $\bullet = \bullet(\textbf{in } s_1 : \sigma, \textbf{in } s_2 : \sigma) \to res : \texttt{bool}$
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} (s_1 = s_2)\}$
Complejidad: $\Theta(equal(s_1, s_2))$
Descripción: función de igualdad de σ'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 eliminación. Además, las claves de los elementos iterados no pueden modificarse nunca, por cuestiones de implementación. Cuando d es modificable, decimos que it es modificable.

Para simplificar la notación, vamos a utilizar clave y significado en lugar de Π_1 y Π_2 cuando utilicemos una tupla(κ, σ).

$\textsc{CrearIt}(\textbf{in } d\colon \texttt{dicc}(\kappa, \sigma)) \to res : \texttt{itDicc}(\kappa, \sigma)$
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res), d)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\Theta(1)$
Descripción: crea un iterador bidireccional del diccionario, de forma tal que HayAnterior evalúe a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).
Aliasing: El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique d sin utilizar las funciones del iterador.

$\textsc{HaySiguiente}(\textbf{in } it\colon \texttt{itDicc}(\kappa, \sigma)) \to res : \texttt{bool}$
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{haySiguiente?}(it)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si en el iterador todavía quedan elementos para avanzar.

$\textsc{HayAnterior}(\textbf{in } it\colon \texttt{itDicc}(\kappa, \sigma)) \to res : \texttt{bool}$
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{hayAnterior?}(it)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si en el iterador todavía quedan elementos para retroceder.

$\textsc{Siguiente}(\textbf{in } it\colon \texttt{itDicc}(\kappa, \sigma)) \to res : \texttt{tupla}(\kappa, \sigma)$

Pre ≡ {HaySiguiente?(it)}
Post ≡ {alias($res =_{\text{obs}}$ Siguiente(it))}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento siguiente del iterador.
Aliasing: res.significado es modificable si y sólo si it es modificable. En cambio, res.clave no es modificable.

SiguienteClave(**in** it: `itDicc(`κ, σ`)`) → res : κ
Pre ≡ {HaySiguiente?(it)}
Post ≡ {alias($res =_{\text{obs}}$ Siguiente(it).clave)}
Complejidad: $\Theta(1)$
Descripción: devuelve la clave del elemento siguiente del iterador.
Aliasing: res no es modficable.

SiguienteSignificado(**in** it: `itDicc(`κ, σ`)`) → res : σ
Pre ≡ {HaySiguiente?(it)}
Post ≡ {alias($res =_{\text{obs}}$ Siguiente(it).significado)}
Complejidad: $\Theta(1)$
Descripción: devuelve el significado del elemento siguiente del iterador.
Aliasing: res es modificable si y sólo si it es modficable.

Anterior(**in** it: `itDicc(`κ, σ`)`) → res : `tupla(clave:` κ`, significado:` σ`)`
Pre ≡ {HayAnterior?(it)}
Post ≡ {alias($res =_{\text{obs}}$ Anterior(it))}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento anterior del iterador.
Aliasing: res.`significado` es modificable si y sólo si it es modificable. En cambio, res.`clave` no es modificable.

AnteriorClave(**in** it: `itDicc(`κ, σ`)`) → res : κ
Pre ≡ {HayAnterior?(it)}
Post ≡ {alias($res =_{\text{obs}}$ Anterior(it).clave)}
Complejidad: $\Theta(1)$
Descripción: devuelve la clave del elemento anterior del iterador.
Aliasing: res no es modficable.

AnteriorSignificado(**in** it: `itDicc(`κ, σ`)`) → res : σ
Pre ≡ {HayAnterior?(it)}
Post ≡ {alias($res =_{\text{obs}}$ Anterior(it).significado)}
Complejidad: $\Theta(1)$
Descripción: devuelve el significado del elemento anterior del iterador.
Aliasing: res es modificable si y sólo si it es modficable.

Avanzar(**in/out** it: `itDicc(`κ, σ`)`)
Pre ≡ {$it = it_0 \land$ HaySiguiente?(it)}
Post ≡ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $\Theta(1)$
Descripción: avanza a la posición siguiente del iterador.

Retroceder(**in/out** it: `itDicc(`κ, σ`)`)
Pre ≡ {$it = it_0 \land$ HayAnterior?(it)}
Post ≡ {$it =_{\text{obs}}$ Retroceder(it_0)}
Complejidad: $\Theta(1)$
Descripción: retrocede a la posición anterior del iterador.

EliminarSiguiente(**in/out** it: `itDicc(`κ, σ`)`)
Pre ≡ {$it = it_0 \land$ HaySiguiente?(it)}
Post ≡ {$it =_{\text{obs}}$ EliminarSiguiente(it_0)}
Complejidad: $\Theta(1)$
Descripción: elimina del diccionario la clave del elemento que se encuentra en la posición siguiente.

EliminarAnterior(**in/out** it: `itDicc(`κ, σ`)`)
Pre ≡ {$it = it_0 \land$ HayAnterior?(it)}

Post $\equiv \{it =_{\text{obs}} \text{EliminarAnterior}(it_0)\}$
Complejidad: $\Theta(1)$
Descripción: elimina del diccionario la clave del elemento que se encuentra en la posición anterior.

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Diccionario Extendido(κ, σ)

extiende Diccionario(κ, σ)

otras operaciones (no exportadas)

esPermutacion? : secu(tupla(κ, σ)) $\times$ dicc(κ, σ) $\longrightarrow$ bool
secuADicc : secu(tupla(κ, σ)) $\longrightarrow$ dicc(κ, σ)

axiomas

esPermutacion?(s, d) $\equiv$ d = secuADicc(s) $\wedge$ #claves(d) = long(s)
secuADicc(s) $\equiv$ **if** vacia?(s) **then** vacio **else** definir(Π_1(prim(s)), Π_2(prim(s)), secuADict(fin(s))) **fi**

Fin TAD

Representación

Representación del diccionario

Hay dos opciones básicas para representar el diccionario lineal, con sus pros y sus contras. La que parece más natural, es representarlo como un conjunto de tuplas sobre secuencia (ver Seccion 9). La ventaja de esta representación es que el invariante de representación y la función de abstracción resultan un poco más naturales. La desventaja es que, como en un conjunto no se pueden modificar los valores, no podríamos modificar el significado de una clave dada. Esto es contrario a lo que queremos. Una opción 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 opción puede resultar viable, pero es un poco molesta.

La representación 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-ésima clave de la lista se asocia al i-ésimo significado. En cierto sentido, estamos definiendo al diccionario como un conjunto de claves y una secuencia de significados. Para no repetir la representación y el codigo del diccionario en el conjunto, vamos a representar al conjunto como un diccionario (ver Sección 9). Si bien esto no parece ser una solución natural, tampoco es tan rara, y nos permite resolver el problema reutilizando la mayoría del codigo.

`dicc`(κ, σ) **se representa con** `dic`

donde `dic` es `tupla`(*claves*: `lista`(κ), *significados*: `lista`(σ))

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

Abs : dicc d $\longrightarrow$ dicc(κ, σ) {Rep(d)}
Abs(d) $\equiv$ **if** vacía?(d.claves) **then** vacío **else** definir(prim(d).claves, prim(d).significado, Abs(fin(d))) **fi**

Representación del iterador

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

`itDicc`(κ, σ) **se representa con** `itDic`

donde `itDic` es `tupla`(*claves*: `itLista`(κ), *significados*: `itLista`(σ))

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

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

Join : secu(α) a $\times$ secu(β) b $\longrightarrow$ secu(tupla(α, β)) {long(a) = long(b)}

$\text{Join}(a, b) \equiv$ **if** vacia?(a) **then** $<>$ **else** $\langle \text{prim}(a), \text{prim}(b)\rangle \bullet \text{Join}(\text{Fin}(a), \text{Fin}(b))$ **fi**

Algoritmos

9. Módulo Conjunto Lineal(α)

El módulo Conjunto Lineal provee un conjunto básico 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 inserción 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 α y $\alpha \times \alpha$ en $\mathbb{N}$, respectivamente).[4]

Interfaz

parámetros formales

géneros α

función $\bullet = \bullet$(**in** $a_1 : \alpha$, **in** $a_2 : \alpha$) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} (a_1 = a_2)\}$
Complejidad: $\Theta(equal(a_1, a_2))$
Descripción: función de igualdad de α's

función COPIAR(**in** $a : \alpha$) $\rightarrow$ *res* : α
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} a\}$
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's

se explica con: CONJ(α), ITERADOR BIDIRECCIONAL MODIFICABLE(α).

géneros: `conj`(α), `itConj`(α).

Operaciones básicas de conjunto

VACÍO() $\rightarrow$ *res* : `conj`(α)
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \emptyset\}$
Complejidad: $\Theta(1)$
Descripción: genera un conjunto vacío.

AGREGAR(**in/out** c: `conj`(α), **in** $a : \alpha$) $\rightarrow$ *res* : `itConj`(α)
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: $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
Descripción: agrega el elemento a al conjunto. Para poder acceder al elemento a en $O(1)$, se devuelve un iterador a la posición de a dentro de c.
Aliasing: el elemento a se agrega por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función ELIMINARSIGUIENTE. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique c sin utilizar las funciones del iterador.

AGREGARRAPIDO(**in/out** c: `conj`(α), **in** $a : \alpha$) $\rightarrow$ *res* : `itConj`(α)
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: $\Theta(copy(a))$
Descripción: agrega el elemento $a \notin c$ al conjunto. Para poder acceder al elemento a en $O(1)$, se devuelve un iterador a la posición de a dentro de c.
Aliasing: el elemento a se agrega por copia. El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función ELIMINARSIGUIENTE. Además, anteriores(res) y siguientes(res) podrían cambiar

[4] Nótese que este es un abuso de notación, ya que no estamos describiendo *copy* y *equal* en función del tamaño de a. A la hora de usarlo, habrá que realizar la traducción.

completamente ante cualquier operación que modifique c sin utilizar las funciones del iterador.

EsVacío?(**in** c: `conj(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \emptyset?(c)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si c esta vacío.

Pertenece?(**in** c: `conj(`α`)`, **in** a: α) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} a \in c)\}$
Complejidad: $\Theta\left(\sum_{a' \in c} equal(a, a')\right)$
Descripción: devuelve `true` si y sólo a pertenece al conjunto.

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

Cardinal(**in** c: `conj(`α`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \#c)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de elementos del conjunto.

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

$\bullet = \bullet$(**in** c_1 : `conj(`α`)`, **in** c_2 : `conj(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} c_1 = c_2\}$
Complejidad: $O\left(\sum_{a_1 \in c_1} \sum_{a_2 \in c_2} equal(a_1, a_2)\right)$.
Descripción: 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 eliminación. Además, los elementos iterados no pueden modificarse, por cuestiones de implementación.

CrearIt(**in** c: `conj(`α`)`) $\rightarrow$ res : `itConj(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), c)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\Theta(1)$
Descripción: crea un iterador bidireccional del conjunto, de forma tal que HayAnterior evalúe a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).
Aliasing: El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique c sin utilizar las funciones del iterador.

HaySiguiente(**in** it: `itConj(`α`)`) $\rightarrow$ res : `bool`

Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{haySiguiente?}(it)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si en el iterador todavía quedan elementos para avanzar.

HayAnterior(**in** it: `itConj(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{hayAnterior?}(it)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si en el iterador todavía quedan elementos para retroceder.

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

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

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

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

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

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

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD Conjunto Extendido(α)

extiende Conjunto(α)

otras operaciones (no exportadas)

esPermutacion? : secu(α) $\times$ conj(α) $\longrightarrow$ bool
secuAConj : secu(α) $\longrightarrow$ conj(α)

axiomas

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

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

Fin TAD

Representación

Representación del Conjunto

En este módulo 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 representación no es la más natural, permite resolver unas cuantas cuestiones sin duplicar codigo. La desventaja aparente es que gastamos memoria para guardar datos inútiles. Sin embargo, los lenguajes de programación actuales permiten resolver este problema de forma más o menos elegante. A nosotros no nos va a importar.

`conj`(α) **se representa con** `dicc`(α`, bool)`

Rep : dicc(α,bool) $\longrightarrow$ bool
Rep(d) $\equiv$ true

Abs : dicc(α,bool) d $\longrightarrow$ conj(α) {Rep(d)}
Abs(d) $\equiv$ claves(d)

Representación del iterador

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

`itConj`(α) **se representa con** `itDicc`(α`, bool)`

Rep : itDicc(α, bool) $\longrightarrow$ bool
Rep(it) $\equiv$ true

Abs : itDicc(α, bool) it $\longrightarrow$ itBi(α) {Rep(it)}
Abs(it) $=_{\text{obs}}$ b: itBi(α) | Anteriores(b) = Π_1(Anteriores(it)) $\wedge$ Siguientes(b) = Π_1(Siguientes(it))

Π_1 : secu(tupla(α, β)) $\longrightarrow$ secu(α)
$\Pi_1(s)$ $\equiv$ **if** vacia?(s) **then** <> **else** Π_1(prim(s)) $\bullet$ Π_1(Fin(s)) **fi**

Algoritmos

10. Módulo Conjunto acotado de naturales

El módulo conjunto acotado de naturales provee un conjunto en el que se pueden insertar únicamente los elementos que se encuentran en un rango $[\ell, r]$ de naturales. La inserción, eliminación 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 también permite eliminar los elementos iterados.

Especificación

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 ℓ × nat r ⟶ conjAcotado $\{\ell \leq r\}$
Ag : nat e × conjAcotado c ⟶ conjAcotado $\{\text{Infimo}(c) \leq e \leq \text{Supremo}(c)\}$

otras operaciones
Rango : conjAcotado ⟶ tupla(nat, nat)

axiomas

$$
\begin{aligned}
\text{Infimo}(\emptyset(\ell, r)) &\equiv \ell \\
\text{Infimo}(\text{Ag}(e, c)) &\equiv \text{Infimo}(c) \\
\text{Supremo}(\emptyset(\ell, r)) &\equiv r \\
\text{Supremo}(\text{Ag}(e, c)) &\equiv \text{Supremo}(c) \\
\text{ConjSuby}(\emptyset(\ell, r)) &\equiv \emptyset \\
\text{ConjSuby}(\text{Ag}(e, c)) &\equiv \text{Ag}(e, \text{ConjSuby}(c)) \\
\text{Rango}(c) &\equiv \langle \text{Infimo}(c), \text{Supremo}(c) \rangle
\end{aligned}
$$

Fin TAD

Interfaz

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

géneros: `conjAcotado`, `itConjAcotado`.

Operaciones básicas de conjunto

Vacío(**in** ℓ: `nat`, **in** r: `nat`) → res : `conjAcotado`
Pre ≡ $\{\ell \leq r\}$
Post ≡ $\{res =_{\text{obs}} \emptyset(\ell, r)\}$
Complejidad: $\Theta(r - \ell)$
Descripción: genera un conjunto vacío con el rango $[\ell, r.]$

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

Infimo(**in** c: `conjAcotado`) → res : `nat`
Pre ≡ {true}
Post ≡ $\{res =_{\text{obs}} \text{Infimo}(c)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el valor mínimo que se puede agregar al conjunto.

Supremo(**in** c: `conjAcotado`) → res : `nat`
Pre ≡ {true}
Post ≡ $\{res =_{\text{obs}} \text{Supremo}(c)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve el valor máximo que se puede agregar al conjunto.

EsVacío?(**in** c: `conjAcotado`) → res : `bool`
Pre ≡ {true}
Post ≡ $\{res =_{\text{obs}} \emptyset?(\text{ConjSuby}(c))\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si c esta vacío.

Pertenece?(**in** c: `conjAcotado`, **in** e: `nat`) → res : `bool`
Pre ≡ {true}
Post ≡ $\{res =_{\text{obs}} e \in \text{ConjSuby}(c)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo 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 $\equiv$ {ConjSuby(c) $=_{\text{obs}}$ ConjSuby(c_0) $\setminus \{e\} \wedge$ Rango(c) $=_{\text{obs}}$ Rango(c_0)}
Complejidad: $\Theta(1)$
Descripción: Elimina a de c, si es que estaba. Observar que no es requerido que e pertenezca al rango de c.

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

Copiar(**in** c: `conjAcotado`) $\rightarrow$ res : `conjAcotado`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} c$}
Complejidad: Θ(Supremo(c) - Infimo(c))
Descripción: genera una copia nueva del conjunto.

$\bullet = \bullet$(**in** c_1 : `conjAcotado`, **in** c_2 : `conjAcotado`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} c_1 = c_2\}$
Complejidad: $\Theta(\text{mín}\{\#c_1, \#c_2\})$.
Descripción: 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 eliminación. Todos los naturales del conjunto son iterados por copia.

CrearIt(**in** c: `conjAcotado`) $\rightarrow$ res : `itConjAcotado`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación?(SecuSuby(res), ConjSuby(c))) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\Theta(1)$
Descripción: crea un iterador bidireccional del conjunto, de forma tal que HayAnterior evalúe a `false` (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente).
Aliasing: El iterador se invalida si y sólo si se elimina el elemento siguiente del iterador sin utilizar la función EliminarSiguiente. Además, anteriores(res) y siguientes(res) podrían cambiar completamente ante cualquier operación que modifique c sin utilizar las funciones del iterador.

HaySiguiente(**in** it: `itConjAcotado`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ haySiguiente?(it)}
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si en el iterador todavía quedan elementos para avanzar.

HayAnterior(**in** it: `itConjAcotado`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ hayAnterior?(it)}
Complejidad: $\Theta(1)$
Descripción: devuelve `true` si y sólo si en el iterador todavía quedan elementos para retroceder.

Siguiente(**in** it: `itConjAcotado`) $\rightarrow$ res : `nat`
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {res $=_{\text{obs}}$ Siguiente(it)}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento siguiente a la posición 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: $\Theta(1)$
Descripción: devuelve el elemento anterior a la posición del iterador.
Aliasing: res se devuelve por copia.

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

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

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

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

Representación

Representación del Conjunto

La idea de este módulo 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 tamaño $r-\ell+1$ de forma tal que $\ell \leq e \leq r$ pertenezca al conjunto si y sólo si $a[e-\ell] = \text{true}$. El inconveniente de esta representación 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 único 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 podría 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 sólo 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 $\wedge$ Supremo(c) = e.infimo + tam(e.pertenencia) $-$ 1 $\wedge$ ConjSuby(c) = e.elementos

Representación del iterador

El iterador del conjunto acotado es simplemente un iterador del conjunto *elementos*, ya que con éste recorremos todos los elementos, más 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)`)

$\text{Rep} : \text{itCA} \longrightarrow \text{bool}$
$\text{Rep}(it) \equiv \text{true} \iff \text{Rep}(*it.\text{conj}) \land \text{EsPermutacion}(\text{SecuSuby}(it.\text{iter}), it.\text{conj}\rightarrow\text{elementos})$

$\text{Abs} : \text{itCA } it \longrightarrow \text{itBi(nat)}$ $\{\text{Rep}(it)\}$
$\text{Abs}(it) \equiv it.\text{elementos}$

Algoritmos