

Apunte de Módulos Básicos (v. 0.3α)

Algoritmos y Estructuras de Datos II, DC, UBA.

1^{er} 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. Modulo Mapa	5
5. Módulo Pila(α)	12
6. Módulo Cola(α)	13
7. Módulo Vector(α)	15
8. Módulo Diccionario Lineal(κ, σ)	18
9. Módulo Conjunto Lineal(α)	23
10. Módulo Conjunto acotado de naturales	26

1. Introducci3n

El presente documento describe varios m3dulos que se pueden utilizar para realizar el TP de dise1o. Adem1s, sirve como ejemplo de qu3 se espera del TP de dise1o, y muestra algunas tcnicas que podr3an ser 3tiles a la hora de desarrollar nuevos m3dulos.

Antes de introducir los m3dulos, se especifican los tipos de iteradores que se van a utilizar. Esta especificaci3n es auxiliar, para simplificar las precondiciones y postcondiciones de los algoritmos que utilizan iteradores. Luego, se presentan todos los m3dulos, con su interfaz, representaci3n y c3lculos de complejidad.

NOTA: Este apunte no est3 terminado. Adem1s de ser incompleto (faltan los algoritmos y los c3lculos de complejidad de todos los m3dulos), puede tener (mejor dicho, tiene) errores y podr3a sufrir cambios en cualquier momento.

2. TADs para especificar iteradores

En esta secci3n se describen los TADs que utilizamos en la materia para especificar los iteradores. Los mismos no son m3dulos 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 especificaci3n de los iteradores con TADs podr3a evitarse, pero lo incluimos para simplificar la especificaci3n de los m3dulos.

2.1. TAD ITERADOR UNIDIRECCIONAL(α)

El iterador unidireccional permite recorrer los elementos una 3nica vez, avanzando continuamente. Es el tipo de iterador m3s 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 especificaci3n. La idea final es que esta secuencia describa el orden en el que se recorrer3n los elementos de la estructura, i.e., esta secuencia es una “permutaci3n” de la estructura iterada.

TAD ITERADOR UNIDIRECCIONAL(α)

par3metros formales

g3neros α

g3neros $\text{itUni}(\alpha)$

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 b3sicos

Siguientes : $\text{itUni}(\alpha) \rightarrow \text{secu}(\alpha)$

generadores

CrearItUni : $\text{secu}(\alpha) \rightarrow \text{itUni}(\alpha)$

otras operaciones

HayMas? : $\text{itUni}(\alpha) \rightarrow \text{bool}$

Actual : $\text{itUni}(\alpha) \text{ it} \rightarrow \alpha$

Avanzar : $\text{itUni}(\alpha) \text{ it} \rightarrow \text{itUni}(\alpha)$

$\{\text{HayMas?}(it)\}$
 $\{\text{HayMas?}(it)\}$

axiomas

$\text{Siguientes}(\text{CrearItUni}(i)) \equiv i$

$\text{HayMas?}(it) \equiv \neg \text{Vac3a?}(\text{Siguientes}(it))$

$\text{Actual}(it) \equiv \text{Prim}(\text{Siguientes}(it))$

$\text{Avanzar}(it) \equiv \text{CrearItUni}(\text{Fin}(\text{Siguientes}(it)))$

Fin TAD

2.2. TAD ITERADOR UNIDIRECCIONAL MODIFICABLE(α)

El iterador unidireccional modificable es una extensi3n del iterador unidireccional que permite realizar algunas operaciones de modificaci3n 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 cı́mo es el aliasing entre el iterador y el tipo iterado en el mı́dulo correspondiente.

TAD ITERADOR UNIDIRECCIONAL MODIFICABLE(α)

parámetros formales

$\text{gı́l}_{\frac{1}{2}}\text{neros } \alpha$

géneros $\text{itMod}(\alpha)$

igualdad observacional

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

observadores básicos

$\text{Anteriores} : \text{itMod}(\alpha) \rightarrow \text{secu}(\alpha)$

$\text{Sigüientes} : \text{itMod}(\alpha) \rightarrow \text{secu}(\alpha)$

generadores

$\text{CrearItMod} : \text{secu}(\alpha) \times \text{secu}(\alpha) \rightarrow \text{itMod}(\alpha)$

otras operaciones

$\text{SecuSuby} : \text{itMod}(\alpha) \rightarrow \text{secu}(\alpha)$

$\text{HayMas?} : \text{itMod}(\alpha) \rightarrow \text{bool}$

$\text{Actual} : \text{itMod}(\alpha) \text{ } it \rightarrow \alpha$

$\{\text{HayMas?}(it)\}$

$\text{Avanzar} : \text{itMod}(\alpha) \text{ } it \rightarrow \text{itMod}(\alpha)$

$\{\text{HayMas?}(it)\}$

$\text{Eliminar} : \text{itMod}(\alpha) \text{ } it \rightarrow \text{itMod}(\alpha)$

$\{\text{HayMas?}(it)\}$

$\text{Agregar} : \text{itMod}(\alpha) \times \alpha \rightarrow \text{itMod}(\alpha)$

axiomas

$\text{Anteriores}(\text{CrearItMod}(i, d)) \equiv i$

$\text{Sigüientes}(\text{CrearItMod}(i, d)) \equiv d$

$\text{SecuSuby}(it) \equiv \text{Anteriores}(it) \& \text{Sigüientes}(it)$

$\text{HayMas?}(it) \equiv \neg \text{Vacı́a?}(\text{Sigüientes}(it))$

$\text{Actual}(it) \equiv \text{Prim}(\text{Sigüientes}(it))$

$\text{Avanzar}(it) \equiv \text{CrearItMod}(\text{Anteriores}(it) \circ \text{Actual}(it), \text{Fin}(\text{Sigüientes}(it)))$

$\text{Eliminar}(it) \equiv \text{CrearItMod}(\text{Anteriores}(it), \text{Fin}(\text{Sigüientes}(it)))$

$\text{Agregar}(it, a) \equiv \text{CrearItMod}(\text{Anteriores}(it) \circ a, \text{Sigüientes}(it))$

Fin TAD

2.3. ITERADOR BIDIRECCIONAL(α)

El iterador bidireccional es una generalizaciı́n del iterador unidireccional modificable. El mismo permite recorrer los elementos avanzando y retrocediendo. Si bien se podrı́a hacer una versiı́n de iterador bidireccional no modificable, la especificaciı́n de ambas es similar. Cuando se utilice en un mı́dulo que no permita algunas modificaciones, simplemente se puede omitir el diseı́o de las funciones que realizan estas modificaciones (ver e.g., mı́dulo Conjunto Lineal). Por este motivo, optamos sı́lo por la versiı́n modificable.

Para que el iterador bidireccional sea lo mäs simétrico posible, cambiamos la operaciı́n actual por dos: anterior y siguiente. La idea conceptual es pensar que el iterador estı́ posicionado en el medio de dos posiciones, y puede acceder tanto a la anterior como a la siguiente. Obviamente, la implementaciı́n puede diferir de esta visiı́n conceptual.

TAD ITERADOR BIDIRECCIONAL(α)

parámetros formales

$\text{gı́l}_{\frac{1}{2}}\text{neros } \alpha$

géneros $\text{itBi}(\alpha)$

igualdad observacional

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

observadores básicos

$\text{Anteriores} : \text{itBi}(\alpha) \rightarrow \text{secu}(\alpha)$

Siguientes : $\text{itBi}(\alpha) \longrightarrow \text{secu}(\alpha)$

generadores

CrearItBi : $\text{secu}(\alpha) \times \text{secu}(\alpha) \longrightarrow \text{itBi}(\alpha)$

otras operaciones

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

axiomas

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

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 lenguaje 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 lenguaje de primer orden. Este parámetro ϕ involucra 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 válido. 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 $\text{secu}(\alpha)$ y $\text{conj}(\alpha)$, respectivamente. Si escribimos

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

estamos diciendo que s y c comparten la misma memoria de forma tal que cualquier modificación $\frac{1}{2}n$ sobre s afecta a c y viceversa, de forma tal que se satisface $\text{esPermutacion}(s, c)$. 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 $\frac{1}{2}n$ de $c \cup \{a\}$. Notemos que, en particular, s podría $\frac{1}{2}n$ a cambiar a cualquier permutación $\frac{1}{2}n$, salvo que se indique lo contrario. De la misma forma, si se eliminara un elemento a de s , entonces c también se vería $\frac{1}{2}n$ afectado de forma tal que s sea una permutación $\frac{1}{2}n$ de c . En particular, c pasaría $\frac{1}{2}n$ a ser $c \setminus \{a\}$.

Debemos observar que este invariante no es mágico, sino que es una declaración $\frac{1}{2}n$ como cualquier otra, y el programado debe asegurarse que este invariante se cumpla. En particular, en el ejemplo anterior, no deberíamos permitir la inserción $\frac{1}{2}n$ de elementos repetido en s , ya que dejaría $\frac{1}{2}n$ de ser una posible permutación $\frac{1}{2}n$ de un conjunto.

4. Modulo Mapa

Interfaz

se explica con: HABITACION.

generos: mapa.

Operaciones basicas de mapa

NUEVOMAPA(in $n : \text{nat}$) $\rightarrow res : \text{mapa}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{nuevaHab}(n)\}$

Complejidad: $\Theta(n^2)$

Descripción $\frac{1}{2}n$: genera un mapa de tamaño $n \times n$.

OCUPAR(in $m : \text{mapa}$, in $c : \text{tupla}(\text{int}, \text{int})$) $\rightarrow res : \text{mapa}$

Pre $\equiv \{c \in \text{casilleros}(m) \wedge_L \text{libre}(m, c) \wedge \text{alcanzan}(\text{libres}(m) - c, \text{libres}(m) - c, m)\}$

Post $\equiv \{res =_{\text{obs}} \text{ocupar}(c, h)\}$

Complejidad: $\Theta(1)$

Descripción $\frac{1}{2}n$: ocupa una posición del mapa siempre y cuando este no deje de ser conexo.

TAM(in $m : \text{mapa}$) $\rightarrow res : \text{nat}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{tam}(m)\}$

Complejidad: $\Theta(1)$

Descripción $\frac{1}{2}n$: devuelve el tamaño del mapa.

LIBRE(in $m : \text{mapa}$, in $c : \text{tupla}(\text{int}, \text{int})$) $\rightarrow res : \text{bool}$

Pre $\equiv \{c \in \text{casilleros}(m)\}$

Post $\equiv \{res =_{\text{obs}} \text{libre}(c, m)\}$

Complejidad: $\Theta(1)$

Descripción $\frac{1}{2}n$: devuelve si un elemento está ocupado.

Representación $\frac{1}{2}n$

Representación $\frac{1}{2}n$ 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 $\frac{1}{2}n$, su invariante de representación $\frac{1}{2}n$ y su función $\frac{1}{2}n$ de abstracción $\frac{1}{2}n$ son las siguientes.

mapa se representa con m

donde m es $\text{tupla}(\text{tamano} : \text{nat}, \text{casilleros} : \text{vec}(\text{vec}(\text{bool})))$

$\text{Rep} : \text{lst} \rightarrow \text{bool}$

$$\begin{aligned} \text{Rep}(l) &\equiv \text{true} \iff (l.\text{primero} = \text{NULL}) = (l.\text{longitud} = 0) \wedge_L (l.\text{longitud} \neq 0 \Rightarrow_L \\ &\quad \text{Nodo}(l, l.\text{longitud}) = l.\text{primero} \wedge \\ &\quad (\forall i: \text{nat})(\text{Nodo}(l, i) \rightarrow \text{siguiente} = \text{Nodo}(l, i+1) \rightarrow \text{anterior}) \wedge \\ &\quad (\forall i: \text{nat})(1 \leq i < l.\text{longitud} \Rightarrow \text{Nodo}(l, i) \neq l.\text{primero}) \\ \text{Abs} : \text{mapa } m &\longrightarrow \text{hab} \quad \{\text{Rep}(m)\} \\ \text{Abs}(m) &\equiv m.\text{tamano} =_{\text{obs}} \text{tam}(h) \wedge_L (\forall t: \text{tuple}(\text{nat}, \text{nat}))(0 \leq \Pi_1(t), \Pi_2(t) < m.\text{tamano} - 1 \Rightarrow_L \text{libre}(m, t) =_{\text{obs}} \\ &\quad m.\text{casilleros}[\Pi_1(t)][\Pi_2(t)]) \end{aligned}$$

Representación del iterador

El iterador es simplemente un puntero al nodo siguiente. Este puntero apunta a NULL en el caso en que se llega 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, también hay que guardar una referencia a la lista que está siendo iterada. Además, de esta forma podemos saber si el iterador apunta al primero o no.

itLista(α) se representa con iter

donde **iter** es $\text{tupla}(\text{siguiente: puntero(nodo)}, \text{lista: puntero(lst)})$

$\text{Rep} : \text{iter} \longrightarrow \text{bool}$

$\text{Rep}(it) \equiv \text{true} \iff \text{Rep}(*it.\text{lista}) \wedge_L (it.\text{siguiente} = \text{NULL} \vee_L (\exists i: \text{nat})(\text{Nodo}(*it.\text{lista}, i) = it.\text{siguiente}))$

$\text{Abs} : \text{iter } it \longrightarrow \text{itBi}(\alpha)$

$\{\text{Rep}(it)\}$

$\text{Abs}(it) =_{\text{obs}} b: \text{itBi}(\alpha) \mid \text{Siguietes}(b) = \text{Abs}(\text{Sig}(it.\text{lista}, it.\text{siguiente})) \wedge$
 $\text{Anteriores}(b) = \text{Abs}(\text{Ant}(it.\text{lista}, it.\text{siguiente}))$

$\text{Sig} : \text{puntero(lst)} \times \text{puntero(nodo)} \longrightarrow \text{lst}$

$\{\text{Rep}(\langle l, p \rangle)\}$

$\text{Sig}(i, p) \equiv \text{Lst}(p, l \rightarrow \text{longitud} - \text{Pos}(*l, p))$

$\text{Ant} : \text{puntero(lst)} \times \text{puntero(nodo)} \longrightarrow \text{lst}$

$\{\text{Rep}(\langle l, p \rangle)\}$

$\text{Ant}(i, p) \equiv \text{Lst}(\text{if } p = l \rightarrow \text{primero} \text{ then } \text{NULL} \text{ else } l \rightarrow \text{primero} \text{ fi}, \text{Pos}(*l, p))$

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

$\text{Pos} : \text{lst} \times \text{puntero(nodo)} \longrightarrow \text{puntero(nodo)}$

$\{\text{Rep}(\langle l, p \rangle)\}$

$\text{Pos}(l, p) \equiv \text{if } l.\text{primero} = p \vee l.\text{longitud} = 0 \text{ then } 0 \text{ else } 1 + \text{Pos}(\text{FinLst}(l), p) \text{ fi}$

Algoritmos

En esta sección se hace abuso de notación en los cálculos de complejidad 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

vaciar() $\rightarrow res : \text{lst}$

1: $res \leftarrow \langle \text{NULL}, 0 \rangle$

$\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

iAgregarAdelante(in/out $l : \text{lst}$, in $a : \alpha$) $\rightarrow res : iter$

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

Complejidad: $\Theta(\text{copy}(a))$

Justificaci3n: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(\text{copy}(a))$. Aplicando 3lgebra de $\frac{1}{2}$ rdenes:

$$\Theta(1) + \Theta(1) + \Theta(\text{copy}(a)) = \Theta(\text{copy}(a))$$

iAgregarAtras(in/out $l : \text{lst}$, in $a : \alpha$) $\rightarrow res : iter$

1: $it \leftarrow \text{CrearItUlt}(l)$ $\triangleright \Theta(1)$
 2: $\text{AgregarComoSiguiente}(it, a)$ $\triangleright \Theta(\text{copy}(a))$
 3: $res \leftarrow it$ $\triangleright \Theta(1)$

Complejidad: $\Theta(\text{copy}(a))$

Justificaci3n: El algoritmo tiene llamadas a funciones con costo $\Theta(1)$ y $\Theta(\text{copy}(a))$. Aplicando 3lgebra de $\frac{1}{2}$ rdenes: $\Theta(1) + \Theta(\text{copy}(a)) + \Theta(1) = \Theta(\text{copy}(a))$

iEsVac3o(in $l : \text{lst}$) $\rightarrow res : bool$

1: $res \leftarrow (l.primer0 = \text{NULL})$ $\triangleright \Theta(1)$

Complejidad: $\Theta(1)$

iFin(in/out $l : \text{lst}$)

1: $\text{CrearIt}(l).\text{EliminarSiguiente}()$ $\triangleright \Theta(1) + \Theta(1)$

Complejidad: $\Theta(1)$

Justificaci3n: $\Theta(1) + \Theta(1) = \Theta(1)$

iComienzo(in/out $l : \text{lst}$)

1: $\text{CrearItUlt}(l).\text{EliminarAnterior}()$ $\triangleright \Theta(1) + \Theta(1)$

Complejidad: $\Theta(1)$

Justificaci3n: $\Theta(1) + \Theta(1) = \Theta(1)$

iPrimero(in $l : \text{lst}$) $\rightarrow res : \alpha$

1: $res \leftarrow \text{CrearIt}(l).\text{Siguiente}()$ $\triangleright \Theta(1) + \Theta(1)$

Complejidad: $\Theta(1)$

Justificaci3n: $\Theta(1) + \Theta(1) = \Theta(1)$

3ltimo(in $l : \text{lst}$) $\rightarrow res : \alpha$

1: $res \leftarrow \text{CrearItUlt}(l).\text{Anterior}()$ $\triangleright \Theta(1) + \Theta(1)$

Complejidad: $\Theta(1)$

Justificaci3n: $\Theta(1) + \Theta(1) = \Theta(1)$

iLongitud(in $l : \text{lst}$) $\rightarrow res : nat$

1: $res \leftarrow l.longitud$ $\triangleright \Theta(1)$
Complejidad: $\Theta(1)$

•[•](in $l : \text{lst}$, in $i : nat$) $\rightarrow res : \alpha$

1: $it \leftarrow CrearIt(l)$ $\triangleright \Theta(1)$

2: $indice \leftarrow 0$ $\triangleright \Theta(1)$

3: **while** $indice < i$ **do** $\triangleright \Theta(i)$

4: $Avanzar(it)$ $\triangleright \Theta(1)$

5: $indice \leftarrow indice + 1$ $\triangleright \Theta(1)$

6: **end while**

7: $res \leftarrow Siguiente(it)$ $\triangleright \Theta(1)$
Complejidad: $\Theta(i)$
Justificaci $\frac{1}{2}n$: El algoritmo tiene un ciclo que se va a repetir i veces. En cada ciclo se hacen realizan funciones con costo $\Theta(1)$. Aplicando $\frac{1}{2}$ lgebra de $\frac{1}{2}$ lgenes sabemos que el ciclo tiene un costo total del orden $\Theta(i)$. El costo total del algoritmo ser $\frac{1}{2}$ de: $\Theta(1) + \Theta(1) + \Theta(i) * (\Theta(1) + \Theta(1)) + \Theta(1) = \Theta(i)$

iCopiar(in $l : \text{lst}$) $\rightarrow res : \text{lst}$

1: $res \leftarrow Vacía()$ $\triangleright \Theta(1)$

2: $it \leftarrow CrearIt(l)$ $\triangleright \Theta(1)$

3: **while** $HaySiguiente(it)$ **do** $\triangleright \Theta(long(l))$

4: $AgregarAtras(res, Siguiente(it))$ $\triangleright \Theta(copy(Siguiente(it)))$

5: $Avanzar(it)$ $\triangleright \Theta(1)$

6: **end while**
Complejidad: $\Theta\left(\sum_{i=1}^{long(l)} copy(l[i])\right)$
Justificaci $\frac{1}{2}n$: El algoritmo cuenta con un ciclo que se repetirá $\frac{1}{2} long(l)$ veces (recorre la lista entera). Por cada ciclo realiza una copia del elemento, el costo ser $\frac{1}{2}$ el de copiar el elemento. Por lo tanto, el costo total del ciclo ser $\frac{1}{2}$ 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: $\Theta(1) + \Theta(1) + \Theta(long(l)) * (\Theta(copy(Siguiente(it))) + \Theta(1)) = \Theta\left(\sum_{i=1}^{long(l)} copy(l[i])\right)$

•=_i•(in $l_1 : \text{lst}$, in $l_2 : \text{lst}$) $\rightarrow res : bool$

1: $it_1 \leftarrow CrearIt(l_1)$ $\triangleright \Theta(1)$

2: $it_2 \leftarrow CrearIt(l_2)$ $\triangleright \Theta(1)$

3: **while** $HaySiguiente(it_1) \wedge HaySiguiente(it_2) \wedge Siguiente(it_1) = Siguiente(it_2)$ **do** $\triangleright [*]$

4: $Avanzar(it_1) // \Theta(1)$

5: $Avanzar(it_2) // \Theta(1)$

6: **end while**

7: $res \leftarrow \neg(HaySiguiente(it_1) \vee HaySiguiente(it_2))$ $\triangleright \Theta(1) + \Theta(1)$
Complejidad: $\Theta\left(\sum_{i=1}^{\ell} equal(l_1[i], l_2[i])\right)$, donde $\ell = \min\{long(l_1), long(l_2)\}$. $[*]$
Justificaci $\frac{1}{2}n$: $[*]$ 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

```

1: iCrearIt(in  $l$ : lst)  $\rightarrow res$ : iter
2:  $res \leftarrow \langle l.primer, l \rangle$   $\triangleright \Theta(1)$ 
Complejidad:  $\Theta(1)$ 

```

```

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

```

```

1: iHaySiguiente(in  $it$ : iter)  $\rightarrow res$ : bool
2:  $res \leftarrow it.siguiete \neq NULL$   $\triangleright \Theta(1)$ 
Complejidad:  $\Theta(1)$ 

```

```

1: iHayAnterior(in  $it$ : iter)  $\rightarrow res$ : bool
2:  $res \leftarrow it.siguiete \neq (it.lista \rightarrow primero)$   $\triangleright \Theta(1)$ 
Complejidad:  $\Theta(1)$ 

```

```

1: iSiguiente(in  $it$ : iter)  $\rightarrow res$ :  $\alpha$ 
2:  $res \leftarrow (it.siguiete \rightarrow dato)$   $\triangleright \Theta(1)$ 
Complejidad:  $\Theta(1)$ 

```

```

iAnterior(in  $it$ : iter)  $\rightarrow res$ :  $\alpha$ 
1:  $res \leftarrow (SiguieteReal(it) \rightarrow anterior \rightarrow dato)$   $\triangleright \Theta(1)$ 
Complejidad:  $\Theta(1)$ 

```

```

1: iAvanzar(in/out  $it$ : iter)
2:  $it.siguiete \leftarrow (it.siguiete \rightarrow siguiente)$   $\triangleright \Theta(1)$ 
3: if  $it.siguiete = it.lista \rightarrow primero$  then  $\triangleright \Theta(1)$ 
4:    $it.siguiete \leftarrow NULL$ 
5: end if
Complejidad:  $\Theta(1)$ 
Justificación:  $\frac{1}{2}n: \Theta(1) + \Theta(1) = \Theta(1)$ 

```

```

1: iRetroceder(in/out  $it$ : iter)
2:  $it.siguiete \leftarrow (SiguieteReal(it) \rightarrow anterior)$   $\triangleright \Theta(1)$ 
Complejidad:  $\Theta(1)$ 

```

```

1: iEliminarSiguiente(in/out it: iter)
2: puntero(nodo) temp  $\leftarrow$  it.siguiente
3: (tmp  $\rightarrow$  siguiente  $\rightarrow$  anterior)  $\leftarrow$  (tmp  $\rightarrow$  anterior) ▷ Reencadenamos los nodos //  $\Theta(1)$ 
4: (tmp  $\rightarrow$  anterior  $\rightarrow$  siguiente)  $\leftarrow$  (tmp  $\rightarrow$  siguiente)
5: if (tmp  $\rightarrow$  siguiente) = (it.lista  $\rightarrow$  primero) then ▷ Si borramos el último nodo, ya no hay siguiente //  $\Theta(1)$ 
6:   it.siguiente  $\leftarrow$  NULL
7: else ▷ Sino, avanzamos al siguiente //  $\Theta(1)$ 
8:   it.siguiente  $\leftarrow$  (tmp  $\rightarrow$  siguiente)
9: end if
10: if tmp = (it.lista  $\rightarrow$  primero) then ▷ Si borramos el primer nodo, hay que volver a setear el primero //  $\Theta(1)$ 
11:   (it.lista  $\rightarrow$  primero)  $\leftarrow$  it.siguiente
12: end if
13: tmp  $\leftarrow$  NULL ▷ Se libera la memoria ocupada por el nodo //  $\Theta(1)$ 
14: (it.lista  $\rightarrow$  longitud)  $\leftarrow$  (it.lista  $\rightarrow$  longitud) - 1

Complejidad:  $\Theta(1)$ 
Justificación:  $\frac{1}{2}n$ :  $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$ 

```

```

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

Complejidad:  $\Theta(1)$ 
Justificación:  $\frac{1}{2}n$ :  $\Theta(1) + \Theta(1) = \Theta(1)$ 

```

```

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

Complejidad:  $\Theta(1)$ 
Justificación:  $\frac{1}{2}n$ :  $\Theta(1) + \Theta(1) = \Theta(1)$ 

```

```

1: iAgregarComoAnterior(in/out it: iter, in a:  $\alpha$ )
2: puntero(nodo) sig  $\leftarrow$  SiguienteReal(it)
3: puntero(nodo) nuevo  $\leftarrow$  & <a, NULL, NULL> ▷ Reservamos memoria para el nuevo nodo //  $\Theta(1)$ 
4: if sig = NULL then ▷ Asignamos los punteros de acuerdo a si el nodo es el primero o no en la lista circular //  $\Theta(1)$ 
5:   (nuevo  $\rightarrow$  anterior)  $\leftarrow$  nuevo
6:   (nuevo  $\rightarrow$  siguiente)  $\leftarrow$  nuevo
7: else
8:   (nuevo  $\rightarrow$  anterior)  $\leftarrow$  (sig  $\rightarrow$  anterior)
9:   (nuevo  $\rightarrow$  siguiente)  $\leftarrow$  sig
10: end if
11: (nuevo  $\rightarrow$  anterior  $\rightarrow$  siguiente)  $\leftarrow$  nuevo ▷ Reencadenamos los otros nodos //  $\Theta(1)$ 
12: if it.siguiente = (it.lista  $\rightarrow$  primero) then ▷ Cambiamos el primero en caso de que estemos agregando el primero //  $\Theta(1)$ 
13:   (it.lista  $\rightarrow$  primero)  $\leftarrow$  nuevo
14: end if
15: (it.lista  $\rightarrow$  longitud)  $\leftarrow$  (it.lista  $\rightarrow$  longitud) + 1 ▷  $\Theta(1)$ 

Complejidad:  $\Theta(1)$ 
Justificación:  $\frac{1}{2}n$ :  $\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$ 

```

```
iSiguienteReal(in it: iter) → res : puntero(nodo)                                ▷ Esta es una operación  $\frac{1}{2}n$  privada que  
if it.siguiente = NULL then                                                ▷ devuelve el siguiente como lista circular //  $\Theta(1)$   
    res ← (it.lista → siguiente)  
else  
    res ← it.siguiente  
end if  
Complejidad:  $\Theta(1)$ 
```

5. Müldulo Pila(α)

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

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

güneros: $pila(\alpha)$.

VACÍO $(\text{in } a : \alpha) \rightarrow res : pila(\alpha)$
Pre $\equiv \{\}$
Post $\equiv \{res =_{\text{obs}} \text{vacío}(a)\}$
Complejidad: $\Theta(1)$
Descripción: genera una pila vacío a .

APILAR $(\text{in/out } p : pila(\alpha), \text{in } a : \alpha)$
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.

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

TOPE $(\text{in } p : pila(\alpha)) \rightarrow res : \alpha$
Pre $\equiv \{\neg \text{vacío?}(p)\}$
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 $(\text{in/out } p : pila(\alpha)) \rightarrow res : \alpha$
Pre $\equiv \{p =_{\text{obs}} p_0 \wedge \neg \text{vacío?}(p)\}$
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 .

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

¹Nótese 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** $p : \text{pila}(\alpha) \rightarrow res : \text{pila}(\alpha)$)
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} p\}$
Complejidad: $\Theta \left(\sum_{i=1}^t \text{copy}(p[i]) \right) = O \left(t \max_{i=1}^t \text{copy}(p[i]) \right)$, donde $t = \text{tama}\ddot{\text{i}}_{\frac{1}{2}} o(p)$.
Descripci3n: genera una copia nueva de la pila

• (**in** $p_1 : \text{pila}(\alpha), \text{in } p_2 : \text{pila}(\alpha) \rightarrow res : \text{bool}$)
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} p_1 = p_2\}$
Complejidad: $\Theta \left(\sum_{i=1}^t \text{equal}(p_1[i], p_2[i]) \right)$, donde $t = \min\{\text{tama}\ddot{\text{i}}_{\frac{1}{2}} o(p_1), \text{tama}\ddot{\text{i}}_{\frac{1}{2}} o(p_2)\}$.
Descripci3n: compara p_1 y p_2 por igualdad, cuando α posee operaci3n de igualdad.
Requiere: **•** (**in** $a_1 : \alpha, \text{in } a_2 : \alpha \rightarrow res : \text{bool}$)
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} (a_1 = a_2)\}$
Complejidad: $\Theta(\text{equal}(a_1, a_2))$
Descripci3n: funci3n de igualdad de α 's

Especificaci3n de las operaciones auxiliares utilizadas en la interfaz

TAD Pila Extendida(α)

extiende $\text{PILA}(\alpha)$

otras operaciones (no exportadas)

•[**•**] : $\text{pila}(\alpha) \times \text{nat } i \rightarrow \alpha$ $\{i < \text{tama}\ddot{\text{i}}_{\frac{1}{2}} o(p)\}$

axiomas

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

Fin TAD

Representaci3n

El objetivo de este m3dulo es implementar una pila lo m3s 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 3ltimo elemento. En este caso, elegimos que el tope se encuentre en el primer elemento.

pila(α) **se representa con** **lista**(α)

Rep : $\text{lista}(\alpha) \rightarrow \text{bool}$

Rep(l) $\equiv \text{true}$

Abs : $\text{lista}(\alpha) \times l \rightarrow \text{pila}(\alpha)$

$\{\text{Rep}(l)\}$

Abs(l) $\equiv \text{if } \text{vac}\ddot{\text{i}}a?(l) \text{ then } \text{vac}\ddot{\text{i}}_{\frac{1}{2}} a \text{ else } \text{apilar}(\text{prim}(l), \text{Abs}(\text{fin}(l))) \text{ fi}$

Algoritmos

6. M3dulo Cola(α)

El m3dulo Cola provee una cola en la que s3 lo se puede acceder al proximo de la misma. Por este motivo, no incluye iteradores.

Para describir la complejidad de las operaciones, vamos a llamar $\text{copy}(a)$ al costo de copiar el elemento $a \in \alpha$ (i.e., copy es una funci3n de α en \mathbb{N}).²

²N3tese que este es un abuso de notaci3n, ya que no estamos describiendo copy en funci3n del tama3o de a . A la hora de usarlo, habr3 que realizar la traducci3n

Interfaz

pari $\iota_{\frac{1}{2}}$ metros formales
gi $\iota_{\frac{1}{2}}$ neros α
funcii $\iota_{\frac{1}{2}}$ n COPIAR(**in** $a : \alpha$) $\rightarrow res : \alpha$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} a\}$
Complejidad: $\Theta(\text{copy}(a))$
Descripci $\iota_{\frac{1}{2}}$ n: funcii $\iota_{\frac{1}{2}}$ n de copia de α 's

se explica con: COLA(α).

gi $\iota_{\frac{1}{2}}$ neros: cola(α).

VACI $\iota_{\frac{1}{2}}$ A() $\rightarrow res : \text{cola}(\alpha)$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{vacii}_{\frac{1}{2}} a\}$
Complejidad: $\Theta(1)$
Descripci $\iota_{\frac{1}{2}}$ n: genera una cola vaci $\iota_{\frac{1}{2}}$ a.

ENCOLAR(**in/out** $c : \text{cola}(\alpha)$, **in** $a : \alpha$)
Pre $\equiv \{c =_{\text{obs}} c_0\}$
Post $\equiv \{p =_{\text{obs}} \text{encolar}(c, a)\}$
Complejidad: $\Theta(\text{copy}(a))$
Descripci $\iota_{\frac{1}{2}}$ n: encola a a c
Aliasing: el elemento a se encola por copia.

ESVACIA?(**in** $c : \text{cola}(\alpha)$) $\rightarrow res : \text{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{vacia?}(c)\}$
Complejidad: $\Theta(1)$
Descripci $\iota_{\frac{1}{2}}$ n: devuelve **true** si y si $\iota_{\frac{1}{2}}$ lo si la cola es vaci $\iota_{\frac{1}{2}}$ a.

PROXIMO(**in** $c : \text{cola}(\alpha)$) $\rightarrow res : \alpha$
Pre $\equiv \{\text{yaci}_{\frac{1}{2}} a?(c)\}$
Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{proximo}(c))\}$
Complejidad: $\Theta(1)$
Descripci $\iota_{\frac{1}{2}}$ n: devuelve el proximo de la cola.
Aliasing: res es modificable si y si $\iota_{\frac{1}{2}}$ lo si p es modificable.

DESENCOLAR(**in/out** $c : \text{cola}(\alpha)$)
Pre $\equiv \{c =_{\text{obs}} c_0 \wedge \neg \text{yaci}_{\frac{1}{2}} a?(c)\}$
Post $\equiv \{c =_{\text{obs}} \text{desacolar}(c_0)\}$
Complejidad: $\Theta(1)$
Descripci $\iota_{\frac{1}{2}}$ n: desencola el proximo de c .

TAMA $\iota_{\frac{1}{2}}$ O(**in** $c : \text{cola}(\alpha)$) $\rightarrow res : \text{nat}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{tama}_{\frac{1}{2}} o(c)\}$
Complejidad: $\Theta(1)$
Descripci $\iota_{\frac{1}{2}}$ n: devuelve la cantidad de elementos encolados en c .

COPIAR(**in** $c : \text{cola}(\alpha)$) $\rightarrow res : \text{cola}(\alpha)$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} c\}$
Complejidad: $\Theta\left(\sum_{i=1}^t \text{copy}(c[i])\right)$, donde $t = \text{tama}_{\frac{1}{2}} o(c)$
Descripci $\iota_{\frac{1}{2}}$ n: genera una copia nueva de la cola

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

Post $\equiv \{res =_{\text{obs}} c_1 = c_2\}$

Complejidad: $\Theta\left(\sum_{i=1}^t \text{equal}(c_1[i], c_2[i])\right)$, donde $t = \min\{\text{tam}_{\frac{1}{2}}(c_1), \text{tam}_{\frac{1}{2}}(c_2)\}$.

Descripción: $\frac{1}{2}n$: compara c_1 y c_2 por igualdad, cuando α posee operación de igualdad.

Requiere: $\bullet = \bullet(\text{in } a_1 : \alpha, \text{in } a_2 : \alpha) \rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} (a_1 = a_2)\}$

Complejidad: $\Theta(\text{equal}(a_1, a_2))$

Descripción: $\frac{1}{2}n$: 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)

$\bullet[\bullet] : \text{cola}(\alpha) \times \text{nat } i \rightarrow \alpha$ $\{i < \text{tam}_{\frac{1}{2}}(p)\}$

axiomas

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

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.

$\text{cola}(\alpha)$ se representa con $\text{lista}(\alpha)$

$\text{Rep} : \text{lista}(\alpha) \rightarrow \text{bool}$

$\text{Rep}(l) \equiv \text{true}$

$\text{Abs} : \text{lista}(\alpha) \times l \rightarrow \text{cola}(\alpha)$

$\{\text{Rep}(l)\}$

$\text{Abs}(l) \equiv \text{if } \text{vacía?}(l) \text{ then } \text{vacío} \text{ else } \text{encolar}(\text{ult}(l), \text{Abs}(\text{com}(l))) \text{ fi}$

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 mismo 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 $\text{copy}(a)$ 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 $\sum_{i=1}^n \frac{f(j+i)}{n} \rightarrow 1$ cuando $n \rightarrow \infty$, 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
güeros α
función $\text{COPIAR}(\text{in } a : \alpha) \rightarrow \text{res} : \alpha$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} a\}$
Complejidad: $\Theta(\text{copy}(a))$
Descripción: función de copia de α 's.

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

güeros: $\text{vector}(\alpha)$.

VACÍO $() \rightarrow \text{res} : \text{vector}(\alpha)$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{<>}\}$
Complejidad: $\Theta(1)$
Descripción: genera un vector vacío.

AGREGARATRÁS $(\text{in/out } v : \text{vector}(\alpha), \text{in } a : \alpha)$
Pre $\equiv \{v =_{\text{obs}} v_0\}$
Post $\equiv \{v =_{\text{obs}} v_0 \circ a\}$
Complejidad: $\Theta(f(\text{long}(v)) + \text{copy}(a))$
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 $\text{long}(v)$ es potencia de 2.

ESVACÍO $(\text{in } v : \text{vector}(\alpha)) \rightarrow \text{res} : \text{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{vacía?}(v)\}$
Complejidad: $\Theta(1)$
Descripción: devuelve **true** si y sólo si v está vacío.

COMIENZO $(\text{in/out } v : \text{vector}(\alpha))$
Pre $\equiv \{v =_{\text{obs}} v_0 \wedge \neg \text{vacía?}(v)\}$
Post $\equiv \{v =_{\text{obs}} \text{com}(v_0)\}$
Complejidad: $\Theta(1)$
Descripción: elimina el último elemento de v .

TOMARPRIMEROS $(\text{in/out } v : \text{vector}(\alpha), \text{in } n : \text{nat})$
Pre $\equiv \{v =_{\text{obs}} v_0\}$
Post $\equiv \{v =_{\text{obs}} \text{Tomar}(v_0, n)\}$
Complejidad: $\Theta(1)$
Descripción: elimina los últimos $\max\{\text{long}(v) - n, 0\}$ elementos del vector, i.e., se queda con los primeros n elementos del vector.

TIRARULTIMOS $(\text{in/out } v : \text{vector}(\alpha), \text{in } n : \text{nat})$
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 $\max\{\text{long}(v), n\}$ elementos del vector.

ULTIMO $(\text{in } v : \text{vector}(\alpha)) \rightarrow \text{res} : \alpha$
Pre $\equiv \{\neg \text{vacía?}(v)\}$
Post $\equiv \{\text{alias}(\text{res} =_{\text{obs}} \text{ult}(v))\}$

Complejidad: $\Theta(1)$

Descripción: $\frac{1}{2}n$: devuelve el i -ésimo elemento del vector.

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

LONGITUD(in l : vector(α)) $\rightarrow res$: nat

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta(1)$

Descripción: $\frac{1}{2}n$: devuelve la cantidad de elementos que contiene el vector.

•[•](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: $\frac{1}{2}n$: 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: $\Theta(f(\text{long}(v)) + \text{long}(v) - i + \text{copy}(a))$

Descripción: $\frac{1}{2}n$: 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 $\text{long}(v)$ 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: $\frac{1}{2}n$: elimina el elemento que ocupa la posición i de v .

COPIAR(in v : vector(α)) $\rightarrow res$: vector(α)

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta\left(\sum_{i=1}^{\ell} \text{copy}(v[i])\right)$, donde $\ell = \text{long}(v)$.

Descripción: $\frac{1}{2}n$: genera una copia nueva del vector.

• = •(in v_1 : vector(α), in v_2 : vector(α)) $\rightarrow res$: bool

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} v_1 = v_2\}$

Complejidad: $\Theta\left(\sum_{i=1}^{\ell} \text{equal}(v_1[i], v_2[i])\right)$, donde $\ell = \min\{\text{long}(v_1), \text{long}(v_2)\}$.

Descripción: $\frac{1}{2}n$: compara v_1 y v_2 por igualdad, cuando α posee operación de igualdad.

Requiere: **• = •**(in a_1 : α , in a_2 : α) $\rightarrow res$: bool

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} (a_1 = a_2)\}$

Complejidad: $\Theta(\text{equal}(a_1, a_2))$

Descripción: $\frac{1}{2}n$: 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) $s \times \text{nat } i \times \alpha a \rightarrow \text{secu}(\alpha)$

Eliminar : secu(α) $s \times \text{nat } i \rightarrow \text{secu}(\alpha)$

$\{i \leq \text{long}(s)\}$

$\{i < \text{long}(s)\}$

Tomar : $\text{secu}(\alpha) \times \text{nat} \longrightarrow \text{secu}(\alpha)$

otras operaciones (no exportadas)

Tirar : $\text{secu}(\alpha) \times \text{nat} \longrightarrow \text{secu}(\alpha)$

axiomas

Agregar(s, i, a) \equiv (Tomar(n, i) \circ a) & Tirar(n, i)

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

Tomar(s, n) \equiv **if** $n = 0 \vee \text{vacía?}(s)$ **then** $\langle \rangle$ **else** $\text{prim}(s) \bullet \text{Tomar}(\text{fin}(s), n - 1)$ **fi**

Tirar(s, n) \equiv **if** $n = 0 \vee \text{vacía?}(s)$ **then** s **else** Tirar($\text{fin}(s), n - 1$) **fi**

Fin TAD

Representaci3n

La idea de este m3dulo es tener una lista donde el i -ésimo se puede obtener en tiempo $O(1)$. Para esto, necesitamos usar alg3n tipo de acceso aleatorio a los elementos, que se consigue utilizando un arreglo. Adem3s, 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 tama3o 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** \longrightarrow **bool**

Rep(v) \equiv $\text{true} \iff (\exists k: \text{nat})(\text{tam}(v.\text{elementos}) = 2^k \wedge v.\text{longitud} \leq \text{tam}(v.\text{elementos}) \wedge$
 $(\forall i: \text{nat})(0 \leq i < v.\text{longitud} \Rightarrow \text{def?}(v.\text{elementos}, i)) \wedge$
 $(\forall i, j: \text{nat})(0 \leq i < j < v.\text{longitud} \Rightarrow v.\text{elementos}[i] \neq v.\text{elementos}[j]))$

Abs : **vec** $v \longrightarrow \text{secu}(\alpha)$

Abs(v) \equiv **if** $v.\text{longitud} = 0$ **then**

$\langle \rangle$

else

Abs($\langle v.\text{elementos}, v.\text{longitud} - 1 \rangle$) \circ $\ast(v.\text{elementos}[v.\text{longitud} - 1])$

fi

{Rep(v)}

Algoritmos

8. M3dulo Diccionario Lineal(κ, σ)

El m3dulo Diccionario Lineal provee un diccionario b3sico en el que se puede definir, borrar, y testear si una clave est3 definida en tiempo lineal. Cuando ya se sabe que la clave a definir no est3 definida en el diccionario, la definici3n 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

par3metros formales

g3neros κ, σ

funci3n $\bullet \bullet (\text{in } k_1: \kappa, \text{in } k_2: \kappa) \rightarrow \text{res} : \text{bool}$

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta(\text{equal}(k_1, k_2))$

Descripci3n: \bullet **funci3n** de igualdad de κ 's

funci3n $\text{COPIAR}(\text{in } k: \kappa) \rightarrow \text{res} : \kappa$

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta(\text{copy}(k))$

Descripci3n: \bullet **funci3n** de copia de κ 's

³N3tese que este es un abuso de notaci3n, ya que no estamos describiendo *copy* y *equal* en funci3n del tama3o de k . A la hora de usarlo, habr3 que realizar la traducci3n.

funcii $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ COPIAR(**in** $s : \sigma \rightarrow res : \sigma$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} s\}$
Complejidad: $\Theta(\text{copy}(s))$
Descripci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$: funcii $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ de copia de σ 's

se explica con: DICCIONARIO(κ, σ), ITERADOR BIDIRECCIONAL(TUPLA(κ, σ)).

g $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ eros: **dicc**(κ, σ), **itDicc**(κ, σ).

Operaciones b $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{s}$ icas de diccionario

VACI $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{o}$ () $\rightarrow res : \text{dicc}(\kappa, \sigma)$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{vacio}\}$
Complejidad: $\Theta(1)$
Descripci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$: genera un diccionario vaci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{o}$.

DEFINIR(**in/out** $d : \text{dicc}(\kappa, \sigma)$, **in** $k : \kappa$, **in** $s : \sigma \rightarrow res : \text{itDicc}(\kappa, \sigma)$
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{esPermutaci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ (**SecuSuby**(res), d))\}$

Complejidad: $\Theta\left(\sum_{k' \in K} \text{equal}(k, k') + \text{copy}(k) + \text{copy}(s)\right)$, donde $K = \text{claves}(d)$

Descripci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$: define la clave k con el significado s en el diccionario. Retorna un iterador al elemento reci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ agregado.

Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y si $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{l}$ o si se elimina el elemento siguiente del iterador sin utilizar la funcii $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ ELIMINARSIGUIENTE. Adem $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{i}$ s, anteriores(res) y siguientes(res) podri $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{a}$ n cambiar completamente ante cualquier operaci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ que modifique el d sin utilizar las funciones del iterador.

DEFINIRRAPIDO(**in/out** $d : \text{dicc}(\kappa, \sigma)$, **in** $k : \kappa$, **in** $s : \sigma \rightarrow res : \text{itDicc}(\kappa, \sigma)$
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{esPermutaci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ (**SecuSuby**(res), d))\}$

Complejidad: $\Theta(\text{copy}(k) + \text{copy}(s))$

Descripci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$: define la clave $k \notin \text{claves}(d)$ con el significado s en el diccionario. Retorna un iterador al elemento reci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ agregado.

Aliasing: los elementos k y s se definen por copia. El iterador se invalida si y si $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{l}$ o si se elimina el elemento siguiente del iterador sin utilizar la funcii $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ ELIMINARSIGUIENTE. Adem $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{i}$ s, anteriores(res) y siguientes(res) podri $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{a}$ n cambiar completamente ante cualquier operaci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$ que modifique el d sin utilizar las funciones del iterador.

DEFINIDO?(**in** $d : \text{dicc}(\kappa, \sigma)$, **in** $k : \kappa \rightarrow res : \text{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{def?}(d, k)\}$
Complejidad: $\Theta(\sum_{k' \in K} \text{equal}(k, k'))$, donde $K = \text{claves}(d)$
Descripci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$: devuelve **true** si y si $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{l}$ o k est $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{i}$ definido en el diccionario.

SIGNIFICADO(**in** $d : \text{dicc}(\kappa, \sigma)$, **in** $k : \kappa \rightarrow res : \sigma$
Pre $\equiv \{\text{def?}(d, k)\}$
Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Significado}(d, k))\}$
Complejidad: $\Theta(\sum_{k' \in K} \text{equal}(k, k'))$, donde $K = \text{claves}(d)$
Descripci $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{n}$: devuelve el significado de la clave k en d .
Aliasing: res es modificable si y si $_{\frac{1}{2}}^{\frac{1}{2}}\mathbf{l}$ o si d es modificable.

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

Complejidad: $\Theta(\sum_{k' \in K} \text{equal}(k, k'))$, donde $K = \text{claves}(d)$

Descripci3n: $\frac{1}{2}n$: elimina la clave k y su significado de d .

#CLAVES(in $d: \text{dicc}(\kappa, \sigma)$) $\rightarrow res: \text{nat}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \# \text{claves}(d)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve la cantidad de claves del diccionario.

COPIAR(in $d: \text{dicc}(\kappa, \sigma)$) $\rightarrow res: \text{dicc}(\kappa, \sigma)$

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta\left(\sum_{k \in K} (\text{copy}(k) + \text{copy}(\text{significado}(k, d)))\right)$, donde $K = \text{claves}(d)$

Descripci3n: $\frac{1}{2}n$: genera una copia nueva del diccionario.

• = •(in $d_1: \text{dicc}(\kappa, \sigma)$, in $d_2: \text{dicc}(\kappa, \sigma)$) $\rightarrow res: \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} c_1 = c_2\}$

Complejidad: $O\left(\sum_{\substack{k_1 \in K_1 \\ k_2 \in K_2}} \text{equal}(\langle k_1, s_1 \rangle, \langle k_2, s_2 \rangle)\right)$, donde $K_i = \text{claves}(d_i)$ y $s_i = \text{significado}(d_i, k_i)$, $i \in \{1, 2\}$.

Descripci3n: $\frac{1}{2}n$: compara d_1 y d_2 por igualdad, cuando σ posee operaci3n $\frac{1}{2}n$ de igualdad.

Requiere: **• = •**(in $s_1: \sigma$, in $s_2: \sigma$) $\rightarrow res: \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} (s_1 = s_2)\}$

Complejidad: $\Theta(\text{equal}(s_1, s_2))$

Descripci3n: $\frac{1}{2}n$: funci3n $\frac{1}{2}n$ 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 eliminaci3n $\frac{1}{2}n$. Adem3s, las claves de los elementos iterados no pueden modificarse nunca, por cuestiones de implementaci3n $\frac{1}{2}n$. Cuando d es modificable, decimos que it es modificable.

Para simplificar la notaci3n $\frac{1}{2}n$, vamos a utilizar clave y significado en lugar de Π_1 y Π_2 cuando utilicemos una tupla (κ, σ) .

CREARIT(in $d: \text{dicc}(\kappa, \sigma)$) $\rightarrow res: \text{itDicc}(\kappa, \sigma)$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{alias}(\text{esPermutaci3n}(\text{SecuSuby}(res), d)) \wedge \text{vacia?}(\text{Anteriores}(res))\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: crea un iterador bidireccional del diccionario, de forma tal que HAYANTERIOR eval3a a **false** (i.e., que se pueda recorrer los elementos aplicando iterativamente SIGUIENTE).

Aliasing: El iterador se invalida si y s3lo si se elimina el elemento siguiente del iterador sin utilizar la funci3n $\frac{1}{2}n$ ELIMINARSIGUIENTE. Adem3s, $\text{anteriores}(res)$ y $\text{siguientes}(res)$ podr3an cambiar completamente ante cualquier operaci3n $\frac{1}{2}n$ que modifique d sin utilizar las funciones del iterador.

HAYSIGUIENTE(in $it: \text{itDicc}(\kappa, \sigma)$) $\rightarrow res: \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{haySiguiente?}(it)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve **true** si y s3lo si en el iterador todav3a quedan elementos para avanzar.

HAYANTERIOR(in $it: \text{itDicc}(\kappa, \sigma)$) $\rightarrow res: \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{hayAnterior?}(it)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve **true** si y s3lo si en el iterador todav3a quedan elementos para retroceder.

SIGUIENTE(**in** it : **itDicc**(κ, σ)) $\rightarrow res$: **tupla**(κ, σ)

Pre $\equiv \{\text{HaySiguiente?}(it)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Siguiente}(it))\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve el elemento siguiente del iterador.

Aliasing: res .significado es modificable si y s3lo si it es modificable. En cambio, res .clave no es modificable.

SIGUIENTEC clave(**in** it : **itDicc**(κ, σ)) $\rightarrow res$: κ

Pre $\equiv \{\text{HaySiguiente?}(it)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Siguiente}(it).clave)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve la clave del elemento siguiente del iterador.

Aliasing: res no es modificable.

SIGUIENTESIGNIFICADO(**in** it : **itDicc**(κ, σ)) $\rightarrow res$: σ

Pre $\equiv \{\text{HaySiguiente?}(it)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Siguiente}(it).significado)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve el significado del elemento siguiente del iterador.

Aliasing: res es modificable si y s3lo si it es modificable.

ANTERIOR(**in** it : **itDicc**(κ, σ)) $\rightarrow res$: **tupla**(clave: κ , significado: σ)

Pre $\equiv \{\text{HayAnterior?}(it)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Anterior}(it))\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve el elemento anterior del iterador.

Aliasing: res .significado es modificable si y s3lo si it es modificable. En cambio, res .clave no es modificable.

ANTERIORCLAVE(**in** it : **itDicc**(κ, σ)) $\rightarrow res$: κ

Pre $\equiv \{\text{HayAnterior?}(it)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Anterior}(it).clave)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve la clave del elemento anterior del iterador.

Aliasing: res no es modificable.

ANTERIORESIGNIFICADO(**in** it : **itDicc**(κ, σ)) $\rightarrow res$: σ

Pre $\equiv \{\text{HayAnterior?}(it)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{Anterior}(it).significado)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve el significado del elemento anterior del iterador.

Aliasing: res es modificable si y s3lo si it es modificable.

AVANZAR(**in/out** it : **itDicc**(κ, σ))

Pre $\equiv \{it = it_0 \wedge \text{HaySiguiente?}(it)\}$

Post $\equiv \{it =_{\text{obs}} \text{Avanzar}(it_0)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: avanza a la posici3n siguiente del iterador.

RETROCEDER(**in/out** it : **itDicc**(κ, σ))

Pre $\equiv \{it = it_0 \wedge \text{HayAnterior?}(it)\}$

Post $\equiv \{it =_{\text{obs}} \text{Retroceder}(it_0)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: retrocede a la posici3n anterior del iterador.

ELIMINARSIGUIENTE(**in/out** it : **itDicc**(κ, σ))

Pre $\equiv \{it = it_0 \wedge \text{HaySiguiente?}(it)\}$

Post $\equiv \{it =_{\text{obs}} \text{EliminarSiguiente}(it_0)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: elimina del diccionario la clave del elemento que se encuentra en la posici3n siguiente.

ELIMINARANTERIOR(**in/out** it : $\text{itDicc}(\kappa, \sigma)$)

Pre $\equiv \{it = it_0 \wedge \text{HayAnterior?}(it)\}$

Post $\equiv \{it =_{\text{obs}} \text{EliminarAnterior}(it_0)\}$

Complejidad: $\Theta(1)$

Descripción: $\frac{1}{2}n$: elimina del diccionario la clave del elemento que se encuentra en la posición $\frac{1}{2}n$ anterior.

Especificación $\frac{1}{2}n$ de las operaciones auxiliares utilizadas en la interfaz

TAD Diccionario Extendido(κ, σ)

extiende DICCIONARIO(κ, σ)

otras operaciones (no exportadas)

$\text{esPermutacion?} : \text{secu}(\text{tupla}(\kappa, \sigma)) \times \text{dicc}(\kappa, \sigma) \rightarrow \text{bool}$

$\text{secuADicc} : \text{secu}(\text{tupla}(\kappa, \sigma)) \rightarrow \text{dicc}(\kappa, \sigma)$

axiomas

$\text{esPermutacion?}(s, d) \equiv d = \text{secuADicc}(s) \wedge \# \text{claves}(d) = \text{long}(s)$

$\text{secuADicc}(s) \equiv \text{if vacia?}(s) \text{ then vacio else definir}(\Pi_1(\text{prim}(s)), \Pi_2(\text{prim}(s)), \text{secuADict}(\text{fin}(s))) \text{ fi}$

Fin TAD

Representación $\frac{1}{2}n$

Representación $\frac{1}{2}n$ 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 Sección 9). La ventaja de esta representación $\frac{1}{2}n$ es que el invariante de representación $\frac{1}{2}n$ y la función $\frac{1}{2}n$ de abstracción $\frac{1}{2}n$ 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 $\frac{1}{2}n$ 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 $\frac{1}{2}n$ puede resultar viable, pero es un poco molesta.

La representación $\frac{1}{2}n$ 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 sí puede. Además, 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 $\frac{1}{2}n$ y el código del diccionario en el conjunto, vamos a representar al conjunto como un diccionario (ver Sección 9). Si bien esto no parece ser una solución $\frac{1}{2}n$ natural, tampoco es tan rara, y nos permite resolver el problema reutilizando la mayoría $\frac{1}{2}n$ del código.

$\text{dicc}(\kappa, \sigma)$ se representa con dic

donde dic es $\text{tupla}(\text{claves: lista}(\kappa), \text{significados: lista}(\sigma))$

$\text{Rep} : \text{dic} \rightarrow \text{bool}$

$\text{Rep}(d) \equiv \text{true} \iff \# \text{claves}(\text{secuADicc}(d.\text{claves})) = \text{long}(d.\text{claves}) \wedge \text{long}(d.\text{claves}) = \text{long}(d.\text{significados})$

$\text{Abs} : \text{dicc } d \rightarrow \text{dicc}(\kappa, \sigma)$

$\{\text{Rep}(d)\}$

$\text{Abs}(d) \equiv \text{if vacio?}(d.\text{claves}) \text{ then vacio else definir}(\text{prim}(d).\text{claves}, \text{prim}(d).\text{significado}, \text{Abs}(\text{fin}(d))) \text{ fi}$

Representación $\frac{1}{2}n$ 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.

$\text{itDicc}(\kappa, \sigma)$ se representa con itDic

donde itDic es $\text{tupla}(\text{claves: itLista}(\kappa), \text{significados: itLista}(\sigma))$

$\text{Rep} : \text{itDic} \rightarrow \text{bool}$

$\text{Rep}(it) \equiv \text{true} \iff \text{Rep}(\langle \text{SecuSuby}(it.\text{claves}), \text{SecuSuby}(it.\text{significados}) \rangle)$

$\text{Abs} : \text{itDic } it \rightarrow \text{itBi}(\text{tupla}(\kappa, \sigma))$

$\{\text{Rep}(it)\}$

$\text{Abs}(it) \equiv \text{CrearItBi}(\text{Join}(\text{Anteriores}(it.\text{claves}), \text{Anteriores}(it.\text{significados})), \text{Join}(\text{Siguietes}(it.\text{claves}), \text{Siguietes}(it.\text{significados})))$

$$\begin{aligned} \text{Join} &: \text{seccu}(\alpha) \ a \times \text{seccu}(\beta) \ b \longrightarrow \text{seccu}(\text{tupla}(\alpha, \beta)) & \{\text{long}(a) = \text{long}(b)\} \\ \text{Join}(a, b) &\equiv \text{if vacia?}(a) \ \text{then} \ \langle \rangle \ \text{else} \ \langle \text{prim}(a), \text{prim}(b) \rangle \bullet \text{Join}(\text{Fin}(a), \text{Fin}(b)) \ \text{fi} \end{aligned}$$

Algoritmos

9. Módulo Conjunto Lineal(α)

El mĳc₁/2 dulo Conjunto Lineal provee un conjunto bi_c/2sico 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 inserciŃc₁/5n 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

pari; $\frac{1}{2}$ metros formales

$$g_{\frac{1}{2}}^2 \propto \alpha$$
$$\text{funcii}_{\mathcal{L}}^{\frac{1}{2}} \mathbf{n} \bullet = \bullet (\text{in } a_1 : \alpha, \text{in } a_2 : \alpha) \rightarrow res : \text{bool} \quad \text{funcii}_{\mathcal{L}}^{\frac{1}{2}} \mathbf{n} \text{COPIAR}(\text{in } a : \alpha) \rightarrow res : \alpha$$
$$\mathbf{Pre} \equiv \{\text{true}\}$$
$$\mathbf{Pre} \equiv \{\text{true}\}$$
$$\mathbf{Post} \equiv \{res =_{\text{obs}} (a_1 = a_2)\}$$
$$\mathbf{Post} \equiv \{res =_{\text{obs}} a\}$$

Complejidad: $\Theta(equal(a_1, a_2))$

Complejidad: $\Theta(\text{copy}(a))$

Descripcii: $\frac{1}{2}n$: funcii $\frac{1}{2}n$ de egalad de α 's

Descripcii: $\frac{1}{2}n$: funcții; $\frac{1}{2}n$ de copia de α 's

se explica con: $\text{CONJ}(\alpha)$, $\text{ITERADOR BIDIRECCIONAL MODIFICABLE}(\alpha)$.

gi $\frac{1}{2}$ **neros**: $\text{conj}(\alpha)$, $\text{itConj}(\alpha)$.

Operaciones básicas de conjunto

$$\mathrm{VAC\ddot{I}}_{\mathcal{L}}^{\frac{1}{2}\mathrm{O}}() \rightarrow res : \mathrm{conj}(\alpha)$$
$$\text{Pre} \equiv \{\text{true}\}$$
$$\mathbf{Post} \equiv \{res =_{\text{obs}} \emptyset\}$$

Complejidad: $\Theta(1)$

Descripții: $\frac{1}{2}n$: genera un conjunto vaci; $\frac{1}{2}o$.

$$\text{AGREGAR}(\text{in/out } c : \text{conj}(\alpha), \text{in } a : \alpha) \rightarrow \text{res} : \text{itConj}(\alpha)$$
$$\mathbf{Pre} \equiv \{c =_{\text{obs}} c_0\}$$
$$\text{Post} \equiv \{c =_{\text{obs}} Ag(a, c_0) \wedge \text{HaySiguiente}(res) \wedge_L \text{Siguiente}(res) = a \wedge \text{alias}(\text{esPermutacion?}(\text{SecuSuby}(res), c))\}$$

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 $i \cdot \frac{1}{2}n$ de a dentro de c .

Aliasing: el elemento a se agrega por copia. El iterador se invalida si y $\text{si } i_{\frac{1}{2}}$ lo si se elimina el elemento siguiente del iterador sin utilizar la funci $\ddot{o}n$ ELIMINARSIGUIENTE . Adem $\acute{a}s$, anteriores(res) y siguientes(res) podr \acute{a} $i_{\frac{1}{2}}$ an cambiar completamente ante cualquier operaci $\ddot{o}n$ $i_{\frac{1}{2}}$ que modifique c sin utilizar las funciones del iterador.

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

Complejidad: $\Theta(\text{copy}(a))$

⁴Ni₆¹/₂tese que este es un abuso de notaci³i₆¹/₂n, ya que no estamos describiendo *copy* y *equal* en funci³i₆¹/₂n del tama³i₆¹/₂o de *a*. A la hora de usarlo, habri³i₆¹/₂ que realizar la traducci³i₆¹/₂n.

Descripci3n: agrega el elemento $a \notin c$ al conjunto. Para poder acceder al elemento a en $O(1)$, se devuelve un iterador a la posici3n de a dentro de c .

Aliasing: el elemento a se agrega por copia. El iterador se invalida si y s3lo si se elimina el elemento siguiente del iterador sin utilizar la funci3n ELIMINARSIGUIENTE. Adem3s, anteriores(res) y siguientes(res) podr3an cambiar completamente ante cualquier operaci3n que modifique c sin utilizar las funciones del iterador.

ESVAC3O?($\text{in } c : \text{conj}(\alpha)$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \emptyset?(c)\}$

Complejidad: $\Theta(1)$

Descripci3n: devuelve **true** si y s3lo si c esta vac3o.

PERTENECE?($\text{in } c : \text{conj}(\alpha)$, $\text{in } a : \alpha$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta\left(\sum_{a' \in c} \text{equal}(a, a')\right)$

Descripci3n: devuelve **true** si y s3lo si a pertenece al conjunto.

ELIMINAR($\text{in } c : \text{conj}(\alpha)$, $\text{in } a : \alpha$)

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta\left(\sum_{a' \in c} \text{equal}(a, a')\right)$

Descripci3n: elimina a de c , si es que estaba.

CARDINAL($\text{in } c : \text{conj}(\alpha)$) $\rightarrow res : \text{nat}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \#c\}$

Complejidad: $\Theta(1)$

Descripci3n: devuelve la cantidad de elementos del conjunto.

COPIAR($\text{in } c : \text{conj}(\alpha)$) $\rightarrow res : \text{conj}(\alpha)$

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta\left(\sum_{a \in c} \text{copy}(a)\right)$

Descripci3n: genera una copia nueva del conjunto.

$\bullet = \bullet$ ($\text{in } c_1 : \text{conj}(\alpha)$, $\text{in } c_2 : \text{conj}(\alpha)$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} c_1 = c_2\}$

Complejidad: $O\left(\sum_{a_1 \in c_1} \sum_{a_2 \in c_2} \text{equal}(a_1, a_2)\right)$.

Descripci3n: 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 eliminaci3n. Adem3s, los elementos iterados no pueden modificarse, por cuestiones de implementaci3n.

CREARIT($\text{in } c : \text{conj}(\alpha)$) $\rightarrow res : \text{itConj}(\alpha)$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{alias}(\text{esPermutacion?}(\text{SecuSuby}(res), c)) \wedge \text{vac3a?}(\text{Anteriores}(res))\}$

Complejidad: $\Theta(1)$

Descripci3n: crea un iterador bidireccional del conjunto, de forma tal que HAYANTERIOR eval3e a **false** (i.e., que se pueda recorrer los elementos aplicando iterativamente SIGUIENTE).

Aliasing: El iterador se invalida si y si $i_{\frac{1}{2}}$ lo si se elimina el elemento siguiente del iterador sin utilizar la funci3n $i_{\frac{1}{2}}$ ELIMINARSIGUIENTE. Adem3s, $anteriores(res)$ y $siguientes(res)$ podr3n cambiar completamente ante cualquier operaci3n que modifique c sin utilizar las funciones del iterador.

HAYSIGUIENTE(**in** $it: itConj(\alpha)$) $\rightarrow res: bool$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} haySiguiente?(it)\}$

Complejidad: $\Theta(1)$

Descripci3n: $i_{\frac{1}{2}}n$: devuelve **true** si y si $i_{\frac{1}{2}}$ lo si en el iterador todav3a quedan elementos para avanzar.

HAYANTERIOR(**in** $it: itConj(\alpha)$) $\rightarrow res: bool$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} hayAnterior?(it)\}$

Complejidad: $\Theta(1)$

Descripci3n: $i_{\frac{1}{2}}n$: devuelve **true** si y si $i_{\frac{1}{2}}$ lo si en el iterador todav3a quedan elementos para retroceder.

SIGUIENTE(**in** $it: itConj(\alpha)$) $\rightarrow res: \alpha$

Pre $\equiv \{HaySiguiente?(it)\}$

Post $\equiv \{alias(res =_{obs} Siguiente(it))\}$

Complejidad: $\Theta(1)$

Descripci3n: $i_{\frac{1}{2}}n$: devuelve el elemento siguiente a la posici3n $i_{\frac{1}{2}}n$ del iterador.

Aliasing: res no es modificable.

ANTERIOR(**in** $it: itConj(\alpha)$) $\rightarrow res: \alpha$

Pre $\equiv \{HayAnterior?(it)\}$

Post $\equiv \{alias(res =_{obs} Anterior(it))\}$

Complejidad: $\Theta(1)$

Descripci3n: $i_{\frac{1}{2}}n$: devuelve el elemento anterior a la posici3n $i_{\frac{1}{2}}n$ del iterador.

Aliasing: res no es modificable.

AVANZAR(**in/out** $it: itConj(\alpha)$)

Pre $\equiv \{it = it_0 \wedge HaySiguiente?(it)\}$

Post $\equiv \{it =_{obs} Avanzar(it_0)\}$

Complejidad: $\Theta(1)$

Descripci3n: $i_{\frac{1}{2}}n$: Avanza a la posici3n $i_{\frac{1}{2}}n$ siguiente del iterador.

RETROCEDER(**in/out** $it: itConj(\alpha)$)

Pre $\equiv \{it = it_0 \wedge HayAnterior?(it)\}$

Post $\equiv \{it =_{obs} Retroceder(it_0)\}$

Complejidad: $\Theta(1)$

Descripci3n: $i_{\frac{1}{2}}n$: Retrocede a la posici3n $i_{\frac{1}{2}}n$ anterior del iterador.

ELIMINARSIGUIENTE(**in/out** $it: itConj(\alpha)$)

Pre $\equiv \{it = it_0 \wedge HaySiguiente?(it)\}$

Post $\equiv \{it =_{obs} EliminarSiguiente(it_0)\}$

Complejidad: $\Theta(1)$

Descripci3n: $i_{\frac{1}{2}}n$: Elimina de la lista iterada el valor que se encuentra en la posici3n $i_{\frac{1}{2}}n$ siguiente del iterador.

ELIMINARANTERIOR(**in/out** $it: itConj(\alpha)$)

Pre $\equiv \{it = it_0 \wedge HayAnterior?(it)\}$

Post $\equiv \{it =_{obs} EliminarAnterior(it_0)\}$

Complejidad: $\Theta(1)$

Descripci3n: $i_{\frac{1}{2}}n$: Elimina de la lista iterada el valor que se encuentra en la posici3n $i_{\frac{1}{2}}n$ anterior del iterador.

Especificaci3n 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) $\rightarrow bool$

$$\text{secuAConj} : \text{secu}(\alpha) \longrightarrow \text{conj}(\alpha)$$

axiomas

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

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

Fin TAD

Representación del Conjunto

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 código. La desventaja aparente es que gastamos memoria para guardar datos innecesarios. 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 $\text{dicc}(\alpha, \text{bool})$

$$\text{Rep} : \text{dicc}(\alpha, \text{bool}) \longrightarrow \text{bool}$$

$$\text{Rep}(d) \equiv \text{true}$$

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

$$\text{Abs}(d) \equiv \text{claves}(d)$$

$$\{\text{Rep}(d)\}$$

Representación del iterador

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

itConj(α) se representa con $\text{itDicc}(\alpha, \text{bool})$

$$\text{Rep} : \text{itDicc}(\alpha, \text{bool}) \longrightarrow \text{bool}$$

$$\text{Rep}(it) \equiv \text{true}$$

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

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

$$\{\text{Rep}(it)\}$$

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

$$\Pi_1(s) \equiv \text{if vacia?}(s) \text{ then } \langle \rangle \text{ else } \Pi_1(\text{prim}(s)) \bullet \Pi_1(\text{Fin}(s)) \text{ fi}$$

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

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

observadores básicos

Infimo : conjAcotado \rightarrow nat
 Supremo : conjAcotado \rightarrow nat
 ConjSuby : conjAcotado \rightarrow conj(nat)

generadores

\emptyset : nat $\ell \times$ nat $r \rightarrow$ conjAcotado
 Ag : nat $e \times$ conjAcotado $c \rightarrow$ conjAcotado

$\{\ell \leq r\}$
 $\{\text{Infimo}(c) \leq e \leq \text{Supremo}(c)\}$

otras operaciones

Rango : conjAcotado \rightarrow 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($e, \text{ConjSuby}(c)$)
 Rango(c) $\equiv \langle \text{Infimo}(c), \text{Supremo}(c) \rangle$

Fin TAD**Interfaz**

se explica con: CONJUNTO ACOTADO, ITERADOR BIDIRECCIONAL(NAT).

gñ₂¹neros: conjAcotado, itConjAcotado.

Operaciones bñ₂¹sicas de conjunto

VACİ₂¹O(in ℓ : nat, in r : nat) \rightarrow res : conjAcotado

Pre $\equiv \{\ell \leq r\}$

Post $\equiv \{res =_{\text{obs}} \emptyset(\ell, r)\}$

Complejidad: $\Theta(r - \ell)$

Descripcİ₂¹n: genera un conjunto vacİ₂¹o 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 $\equiv \{c =_{\text{obs}} \text{Ag}(e, c_0)\}$

Complejidad: $\Theta(1)$

Descripcİ₂¹n: agrega el elemento e al conjunto.

INFIMO(in c : conjAcotado) \rightarrow res : nat

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{Infimo}(c)\}$

Complejidad: $\Theta(1)$

Descripcİ₂¹n: devuelve el valor mİ₂¹nimo que se puede agregar al conjunto.

SUPREMO(in c : conjAcotado) \rightarrow res : nat

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{Supremo}(c)\}$

Complejidad: $\Theta(1)$

Descripcİ₂¹n: devuelve el valor mİ₂¹ximo que se puede agregar al conjunto.

ESVACİ₂¹O?(in c : conjAcotado) \rightarrow res : bool

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \emptyset?(\text{ConjSuby}(c))\}$

Complejidad: $\Theta(1)$

Descripcİ₂¹n: devuelve true si y sİ₂¹lo si c esta vacİ₂¹o.

PERTENECE?(in c : conjAcotado, in e : nat) \rightarrow res : bool

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} e \in \text{ConjSuby}(c)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve **true** si y s3lo 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 $\equiv \{\text{ConjSuby}(c) =_{\text{obs}} \text{ConjSuby}(c_0) \setminus \{e\} \wedge \text{Rango}(c) =_{\text{obs}} \text{Rango}(c_0)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: 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 $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \# \text{ConjSuby}(c)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: Devuelve la cantidad de elementos del conjunto.

COPIAR(**in** c : conjAcotado) $\rightarrow res$: conjAcotado

Pre $\equiv \{\text{true}\}$

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

Complejidad: $\Theta(\text{Supremo}(c) - \text{Infimo}(c))$

Descripci3n: $\frac{1}{2}n$: genera una copia nueva del conjunto.

= (**in** c_1 : conjAcotado, **in** c_2 : conjAcotado) $\rightarrow res$: bool

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} c_1 = c_2\}$

Complejidad: $\Theta(\min\{\#c_1, \#c_2\})$.

Descripci3n: $\frac{1}{2}n$: 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 eliminaci3n. Todos los naturales del conjunto son iterados por copia.

CREARIT(**in** c : conjAcotado) $\rightarrow res$: itConjAcotado

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{alias}(\text{esPermutaci3n}(\text{SecuSuby}(res), \text{ConjSuby}(c))) \wedge \text{vac3a}(\text{Anteriores}(res))\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: crea un iterador bidireccional del conjunto, de forma tal que HAYANTERIOR eval3a a **false** (i.e., que se pueda recorrer los elementos aplicando iterativamente SIGUIENTE).

Aliasing: El iterador se invalida si y s3lo si se elimina el elemento siguiente del iterador sin utilizar la funci3n ELIMINARSIGUIENTE. Adem3s, $\text{anteriores}(res)$ y $\text{siguientes}(res)$ podr3an cambiar completamente ante cualquier operaci3n que modifique c sin utilizar las funciones del iterador.

HAYSIGUIENTE(**in** it : itConjAcotado) $\rightarrow res$: bool

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{haySiguiente?}(it)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve **true** si y s3lo si en el iterador todav3a quedan elementos para avanzar.

HAYANTERIOR(**in** it : itConjAcotado) $\rightarrow res$: bool

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{hayAnterior?}(it)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve **true** si y s3lo si en el iterador todav3a quedan elementos para retroceder.

SIGUIENTE(**in** it : itConjAcotado) $\rightarrow res$: nat

Pre $\equiv \{\text{HaySiguiente?}(it)\}$

Post $\equiv \{res =_{\text{obs}} \text{Siguiente}(it)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve el elemento siguiente a la posici3n del iterador.

Aliasing: *res* se devuelve por copia.

ANTERIOR(**in** *it*: itConjAcotado) \rightarrow *res* : nat

Pre $\equiv \{\text{HayAnterior?}(it)\}$

Post $\equiv \{res =_{\text{obs}} \text{Anterior}(it)\}$

Complejidad: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: devuelve el elemento anterior a la posici3n $\frac{1}{2}n$ 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: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: Avanza a la posici3n $\frac{1}{2}n$ 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: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: Retrocede a la posici3n $\frac{1}{2}n$ 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: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: Elimina del conjunto el elemento que se encuentra en la posici3n $\frac{1}{2}n$ 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: $\Theta(1)$

Descripci3n: $\frac{1}{2}n$: Elimina del conjunto el elemento que se encuentra en la posici3n $\frac{1}{2}n$ anterior.

Representaci3n $\frac{1}{2}n$

Representaci3n $\frac{1}{2}n$ del Conjunto

La idea de este m3dulo 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 tama3o $r - \ell + 1$ de forma tal que $\ell \leq e \leq r$ pertenezca al conjunto si y s3lo si $a[e - \ell] = \text{true}$. El inconveniente de esta representaci3n $\frac{1}{2}n$ 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 3nico 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 podr3a ser un iterador. Luego, en *a* vamos a tener, adem3s 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 s3lo 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* \rightarrow bool

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

Abs : *ca e* \rightarrow conjAcotado

{*Rep*(*e*)}

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

Representación del iterador

El iterador del conjunto acotado es simplemente un iterador del conjunto *elementos*, ya que con $i_{\frac{1}{2}}$ ste 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 $\text{tupla}(\text{iter}: \text{itConj}(\text{nat}), \text{conj}: \text{puntero}(\text{ca}))$

$\text{Rep} : \text{itCA} \longrightarrow \text{bool}$

$\text{Rep}(it) \equiv \text{true} \iff \text{Rep}(*it.\text{conj}) \wedge \text{EsPermutacion}(\text{SecuSuby}(it.\text{iter}), it.\text{conj} \rightarrow \text{elementos})$

$\text{Abs} : \text{itCA } it \longrightarrow \text{itBi}(\text{nat})$

$\text{Abs}(it) \equiv it.\text{elementos}$

$\{\text{Rep}(it)\}$

Algoritmos