Algoritmos y Estructuras de Datos II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Nº2

Grupo 19

Integrante	LU	Correo electrónico
Basso, Juan Cruz	627/14	jcbasso95@gmail.com
Bohe, Brian	706/14	brianbohe@gmail.com
Figari, Francisco	719/14	francisco.figari@hotmail.com
Mariotti, Ignacio	651/14	nacho692@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Nociones previas

- Para las complejidades se utilizan las definiciones del contexto de uso y se agregan algunas otras
  - $|n_m|$ es el nombre mas largo de estudiantes y hippies
  - $|h_m|$ es el nombre mas largo de hippies
  - $|e_m|$ es el nombre mas largo de estudiantes
  - N_a es la cantidad de agentes
  - N_h es la cantidad de hippies
  - N_e es la cantidad de estudiantes
- Algunas consideraciones de tipos son
  - *direccion* es un enumerado izq,der,arriba,abajo
  - *placa* es nat
  - *agente* es nat
  - *nombre* es string
- Al no haber un modulo de arreglos explicito en modulos basicos, pueden haber diferencias triviales en los diferentes algoritmos
- Se asume que el módulo nat tiene las siguientes funciones
  - MAX(**in** a : `nat`, **in** b: `nat`) $\rightarrow$ res : `nat`
    Pre $\equiv$ {true}
    Post $\equiv$ {res $=_{\text{obs}}$ **if** a >b **then** a **else** b **fi**}
    Complejidad: $\mathcal{O}(1)$
  - MIN(**in** a : `nat`, **in** b: `nat`) $\rightarrow$ res : `nat`
    Pre $\equiv$ {true}
    Post $\equiv$ {res $=_{\text{obs}}$ **if** a <b **then** a **else** b **fi**}
    Complejidad: $\mathcal{O}(1)$
  - MOD(**in** a : `nat`, **in** m: `nat`) $\rightarrow$ res : `nat`
    Pre $\equiv$ {true}
    Post $\equiv$ {res $=_{\text{obs}}$ modulo(a, m)}
    Complejidad: $\mathcal{O}(1)$
- Se asume que el módulo bool tiene la siguiente función
  - β(**in** b: `bool`) $\rightarrow$ res : `nat`
    Pre $\equiv$ {true}
    Post $\equiv$ {res $=_{\text{obs}}$ **if** b **then** 1 **else** 0 **fi**}
    Complejidad: $\mathcal{O}(1)$

modulo : nat $\times$ nat $\longrightarrow$ nat
modulo(a,m) $\equiv$ **if** $a < m$ **then** a **else** modulo($a - m$) **fi**

2. Modulos

2.1. Modulo Campus Seguro

Se sigue el siguiente orden para las funciones que proveen los movimientos de estudiantes, agentes o hippies.

1. Apariciones y Movimientos
2. Enhippizacion (Estudiantes en hippies)
3. Educacion (Hippies en estudiantes)

4. Premios de capturas de hippies
5. Captura de hippies
6. Sanciones relacionadas con el evento

Interfaz

se explica con: Campus Seguro.

géneros: `campusSeg`.

Operaciones básicas de campusSeg

Campus(**in** c: `campusSeg`) $\rightarrow$ res : `campus`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$campus(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el campus de Campus Seguro
Aliasing: El campus se devuelve por referencia

Estudiantes(**in** c: `campusSeg`) $\rightarrow$ res : `itClavesDiccN(nombre)`
Pre $\equiv$ {true}
Post $\equiv$ {esPermutacion(Siguientes(res),secuenciarConj(estudiantes(d))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve Iterador no modificable de todos los estudiantes actualmente en el campus
Aliasing: Si hay un cambio en los estudiantes, el iterador puede quedar invalidado

Hippies(**in** c: `campusSeg`) $\rightarrow$ res : `itClavesDiccN(nombre)`
Pre $\equiv$ {true}
Post $\equiv$ {esPermutacion(Siguientes(res),secuenciarConj(hippies(d))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve Iterador no modificable de todos los hippies actualmente en el campus
Aliasing: Si hay un cambio en los hippies, el iterador puede quedar invalidado

Agentes(**in** c: `campusSeg`) $\rightarrow$ res : `itClavesDiccLog(nat)`
Pre $\equiv$ {true}
Post $\equiv$ {esPermutacion(Siguientes(res),secuenciarConj(agentes(c)))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve Iterador no modificable de todos los agentes actualmente en el campus
Aliasing: Si hay un cambio en el diccionario, se va a ver reflejado en el iterador

PosEstudianteYHippie(**in** c: `campusSeg`, **in** n: `nombre`) $\rightarrow$ res : `posicion`
Pre $\equiv$ {$n \in$ estudiantes(c)$\cup$hippies(c)}
Post $\equiv$ {$res =_{\text{obs}}$posEstudianteYHippie(n,c)}
Complejidad: $\mathcal{O}(|n_m|)$ con siendo $|n_m|$ la longitud del nombre más largo de hippies y estudiantes
Descripción: Devuelve la posicion del estudiante o hippie

PosAgente(**in** c: `campusSeg`, **in** p: `placa`) $\rightarrow$ res : `posicion`
Pre $\equiv$ {$pl \in$ agentes(c)}
Post $\equiv$ {$res =_{\text{obs}}$ posAgente(pl,c)}
Complejidad: $\mathcal{O}(1)$ en caso promedio
Descripción: Devuelve la posicion del agente

CantSanciones(**in** c: `campusSeg`, **in** pl: `placa`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$pl \in$ agentes(c)}
Post $\equiv$ {$res =_{\text{obs}}$ cantSanciones(pl,c)}
Complejidad: $\mathcal{O}(1)$ en caso promedio
Descripción: Devuelve la cantidad de sanciones del agente

CantHippiesAtrapados(**in** c: `campusSeg`, **in** pl: `placa`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$pl \in$ agentes(c)}
Post $\equiv$ {$res =_{\text{obs}}$ cantHippiesAtrapados(pl,c)}
Complejidad: $\mathcal{O}(1)$ en caso promedio
Descripción: Devuelve la cantidad de hippies que atrapo el agente

ComenzarRastrillaje(**in** c: `campus`, **in** d: `dicc(agente,posicion)`) $\rightarrow$ res : `campusSeg`
Pre $\equiv$ {($\forall a$: agente)def?(a,d)$\Rightarrow_{\text{L}}$ posValida?(obtener(a,d))$\wedge\neg$ocupada?(obtener(a,d),c)))
$\wedge$($\forall a_0, a_1$: agente)((def?(a_0,d)$\wedge$def?(a_1,d)$\wedge a_0 \neq a_1$)$\Rightarrow_{\text{L}}$obtener(a_0,d)$\neq$obtener(a_1,d))}
Post $\equiv$ {$res =_{\text{obs}}$comenzarRastrillaje(c,d)}
Complejidad: $\mathcal{O}(h * a + n^2)$ + $\mathcal{O}(n)$ prom
Descripción: Inicializa toda la estructura en funcion de los datos de entrada

IngresarEstudiante(**in/out** c: `campusSeg`, **in** n: `nombre`, **in** p: `posicion`)
Pre $\equiv$ {$c = c_0 \wedge n \notin$ (estudiantes(c)$\cup$hippies(c))$\wedge$esIngreso(p,campus(c))$\wedge\neg$ estaOcupada?(p,c)}
Post $\equiv$ {$c =$ ingresarEstudiante(n,p,c_0)}
Complejidad: $\mathcal{O}(|n_m|)$

IngresarHippie(**in/out** c: `campusSeg`, **in** n: `nombre`, **in** p: `posicion`)
Pre $\equiv$ {$c = c_0 \wedge n \notin$ (estudiantes(c)$\cup$hippies(c))$\wedge$esIngreso(p,campus(c))$\wedge\neg$ estaOcupada?(p,c)}
Post $\equiv$ {$c =$ ingresarHippie(n,p,c_0)}
Complejidad: $\mathcal{O}(|n_m|)$

MoverEstudiante(**in/out** c: `campusSeg`, **in** n: `nombre`, **in** d: `direccion`)
Pre $\equiv$ {$c = c_0 \wedge n \in$ estudiantes(c)$\wedge$(seRetira(n,d,c)$\vee$
(posValida(proxPosicion(posEstudianteYHippie(n,c), d,campus(c)),campus(c))$\wedge$
$\neg$estaOcupada?(proxPosicion(posEstudianteYHippie(n,c),d,campus(c)),c)))}
Post $\equiv$ {$c =$ moverEstudiante(n,d,c_0)}
Complejidad: $\mathcal{O}(|n_m|)$

MoverHippie(**in/out** c: `campusSeg`, **in** n: `nombre`)
Pre $\equiv$ {$c = c_0 \wedge n \in$ hippies(c)$\wedge\neg$todasOcupadas?(vecinos(posEstudianteYHippie(n,c),campus(c)),c)}
Post $\equiv$ {$c =$ moverHippie(n,c_0)}
Complejidad: $\mathcal{O}(|n_m|) + \mathcal{O}(N_e)$

MoverAgente(**in/out** c: `campusSeg`, **in** a: `agente`)
Pre $\equiv$ {$c = c_0 \wedge$
$a \in$ agentes(c)$\wedge_{\text{L}}$ cantSanciones(a,c) $\leq 3 \wedge \neg$todasOcupadas?(vecinos(posAgente(a,c),campus(c)),c) }
Post $\equiv$ {$c =$ moverAgente(a,c_0)}
Complejidad: $\mathcal{O}(|n_m|) + \mathcal{O}(logN_a) + \mathcal{O}(N_h)$

MasVigilante(**in** c: `campusSeg`) $\rightarrow$ res : `placa`
Pre $\equiv$ {$\#Agentes(c) \geq 1$}
Post $\equiv$ {$res =_{\text{obs}}$masVigilante(c)}
Complejidad: $\mathcal{O}(1)$

ConMismasSanciones(**in** c: `campusSeg`, **in** a: `agente`) $\rightarrow$ res : `conj(placa)`
Pre $\equiv$ {$a \in Agentes(c)$}
Post $\equiv$ {$res =_{\text{obs}}$ conMismasSanciones(a,c)}
Complejidad: $\mathcal{O}(1)$
Descripción: El conjunto se devuelve por referencia y no se puede modificar

ConKSanciones(**in/out** c: `campusSeg`, **in** k: `nat`) $\rightarrow$ res : `conj(placa)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ conKsanciones(k,c)}
Complejidad: $\mathcal{O}(N_a)$ o $\mathcal{O}(logN_a)$ dependiendo si hubo cambios en las sanciones
Descripción: El conjunto se devuelve por referencia y no se puede modificar

CantHippies(**in** c: `campusSeg`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ cantHippies(c)}

Complejidad: $\mathcal{O}(1)$

CantEstudiantes(**in** c: `campusSeg`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ cantEstudiantes(c)}
Complejidad: $\mathcal{O}(1)$

Representación

Representación de campusSeg

`campusSeg` **se representa con** `estr`

donde `estr` es `tupla`(*hippies*: `diccNombres(nombre,posicion)`, *estudiantes*: `diccNombres(nombre,posicion)`
, *agentes*: `diccLog(nat,datosAg)`
, *sanc*: `sanciones`
, *agentesRapido*: `diccH(nat,itDiccLog(nat,posicion))`
, *pos*: `posiciones` , *masVigilante*: `tupla(pl: placa, premios: nat)`
, *cp*: `campus`)
donde `sanciones` es `tupla`(*lista*: `lista(tupla(c : nat,s :conj(placa)))` , *rapidas*: `diccLog(nat, con(placa))` , *actualizar*: `bool`)
donde `posiciones` es `tupla`(*mAg*: `matriz(tupla(def: bool,datos: itDiccLog(nat, datosAg))`
, *mH*: `matriz(tupla(def: bool,datos: itDiccN(nombre, posicion))`
, *mE*: `matriz(tupla(def: bool,datos: itDiccN(nombre, posicion))`)
donde `datosAg` es `tupla`(*pl*: `placa` , *sanciones*: `nat` , *premios*: `nat` , *pos*: `posicion`, *kSanc*: `itLista(tupla(nat, conj(placa))`, *mismasSanc*: `itconj(placa)`)

Aclaraciones

Dada la complejidad de la estructura, el invariante puede ser dificil de seguir, por lo que a continuacion se encuentra detallado de manera informal las lineas que se siguieron al hacer el rep.

1. Las claves de hippies y estudiantes intersecan vacio.
2. No hay posiciones iguales en estudiantes, hippies y agentes y todas se encuentran en rango.
3. Las placas de agentes y agentesRapido son iguales y, a igual placa, el siguiente del iterador de agentesRapido es el agente en agentes y la secuencia subyacente del iterador es permutacion de la secuencia de tuplas del diccionario de agentes.
4. Los conjuntos de la lista de sanciones son subconjuntos del total de las placas de agentes. Una placa esta en uno de estos conjuntos si y sólo si el agente correspondiente tiene tantas sanciones como el valor correspondiente a ese conjunto.
5. La lista de sanciones esta ordenada por su primer elemento de la tupla estrictamente
6. Si actualizar esta en false, las tuplas de la lista son tuplas (clave, significado) del diccionario logaritmico y vice versa
7. Para todos los datosAg de agentes, la secuencia subyacente del iterador kSanc es igual a una secuencia ordenada de sanciones y el siguiente primer elemento de la tupla del iterador es la cantidad de sanciones del datosAg.
8. Para todos los datosAg de agentes, la secuencia subyacente del iterador mismasSanc es permutacion del conjunto de placas de la lista de sanciones donde se encuentra su propia placa, y el siguiente del iterador es su misma placa.
9. filas+1 y columnas+1 de cp (campus) es igual al alto y ancho de las matrices de pos.
10. mAg, el booleano es true si y solo si hay un agente en placasPos en esa posicion.
    Si es true, la siguiente clave del iterador es el agente en esa posicion y la secuencia subyacente es permutacion de los agentes en agentes.
11. mH, el booleano es true si y solo si hay un hippie en esa posicion.
    Si es true, la siguiente clave del iterador es el hippie en esa posicion y la secuencia subyacente es permutacion de los hippies en hippies.

12. mE, el booleano es true si y solo si hay un estudiante en esa posicion.
    Si es true,la siguiente clave del iterador es el estudiante en esa posicion y la secuencia subyacente es permutacion de los estudiantes en estudiantes.

13. El mas vigilante es el de mayor sanciones de los agentes y, entre ellos, el de menor placa.

Para facilitar la lectura se renombra las matrices e.pos.mXX a mXX

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$

1. claves(e.hippies) $\cap$ claves(e.estudiantes) $= \emptyset$

2. $\wedge$ ($\forall p$: posicion)(#(p,armarMultiPos(e)) ≥ 1 $\Rightarrow$
   #(p,armarMultiPos(e)) $= 1 \wedge$ posValida(p,e.cp) $\wedge_{\text{L}}$ $\neg$ocupada?(p,e.cp))

3. $\wedge$ (claves(e.agentes) = claves(e.agentesRapido) $\wedge_{\text{L}}$ ($\forall p$: placa) (def?(p,e.agentes) $\Rightarrow_{\text{L}}$
   π_1(Siguiente(obtener(p,e.agentesRapido))) = obtener(p,e.agentes))
   $\wedge$ esPermutacion(SecuSuby(obtener(p,e.agentesRapido)),secuenciarDic(e.agentes)))

4. $\wedge$ ($\forall c$: nat)($\forall s$: conj(placa)) (esta?($\langle c,s \rangle$,e.sanciones.lista) $\Rightarrow$
   ($s \subseteq$ claves(e.agentes) $\wedge$ $\neg\emptyset$?(s)
   $\wedge_{\text{L}}$ ($\forall pl$:placa) $pl \in s \iff$ obtener(pl,e.agentes).sanciones=$c \wedge$ obtener(pl,e.agentes).pl = pl))

5. ($\forall i$: nat)($i \leq$longitud(e.lista)-2)e.sanciones.lista[i].c $<e$.sanciones.lista[$i+1$].c

6. $\neg e$.samnciones.actualizar $\Rightarrow$
   ($\forall c$: nat)($\forall s$:conj(nat))(esta?($\langle$c,s$\rangle$),e.sanciones.lista) $\iff$ def?(c,e.sanciones.rapidas)
   $\wedge_{\text{L}} s$ = obtener(c,e.sanciones.rapidas)

7. $\wedge$ ($\forall p$: placa) def?(p,e.agentes) $\Rightarrow_{\text{L}}$
   SecuSuby(obtener(p,e.agentes).kSanc) = ordenarπ_1(e.sanciones.lista)
   $\wedge$ π_1(Siguiente(obtener(p,e.agentes).kSanc)) = obtener(p,e.agentes).pl

8. $\wedge$ ($\forall c$: nat)($\forall s$: conj(placa))($\forall pl$: placa)
   (esta?($\langle$c,s$\rangle$),e.sanciones.lista) $\wedge$ ($pl \in s \wedge$ def?(pl,e.agentes))) $\Rightarrow_{\text{L}}$
   esPermutacion(secuenciarConj(s),SecuSuby(obtener(pl,e.agentes).mismasSanc))
   $\wedge$ siguiente(obtener(pl,e.agentes).mismasSanc) = pl

9. $\wedge$ Ancho(mAg) = columnas(e.campus) + 1 $\wedge$ Alto(mAg) = filas(e.campus) + 1
   $\wedge$ Ancho(mH) = Ancho(mAg) $\wedge$ Alto(mH) = Alto(mAg)
   $\wedge$ Ancho(mE) = Ancho(mH) $\wedge$ Alto(mE) = Alto(mH)

10. $\wedge$ ($\forall i$: nat) ($\forall j$: nat) $0 < i <$ Ancho(mAg) $\wedge$ $0 < j <$ Alto(mAg) $\Rightarrow_{\text{L}}$
    (π_1(Valor(mAg,i,j)) $\iff$
    ($\exists p$: placa) (def?(p,e.agentes) $\wedge_{\text{L}}$ (obtener(p,e.agentes).pos = $\langle i, j \rangle \wedge$
    esPermutacion(SecuSuby(π_2(Valor(mAg,i,j))), secuenciarDic(e.agentes))
    $\wedge$ π_1(Siguiente(π_2(Valor(mAg,i,j))) = obtener(p,e.agentes).pl))

11. $\wedge$ ($\forall i$: nat) ($\forall j$: nat) $0 < i <$ Ancho(mH) $\wedge$ $0 < j <$ Alto(mH) $\Rightarrow_{\text{L}}$
    (π_1(Valor(mH,i,j)) $\iff$
    ($\exists n$: nombre)(def?(n,e.hippies) $\wedge_{\text{L}}$
    obtener(n,e.hippies) = $\langle i, j \rangle \wedge$ π_1(Siguiente(π_2(Valor(mH,i,j)))) = $n \wedge$
    esPermutacion(SecuSuby(π_2(Valor(mH,i,j)),secuenciarDic(e.hippies)))

12. $\wedge$ ($\forall i$: nat) ($\forall j$: nat) $0 < i <$ Ancho(mE) $\wedge$ $0 < j <$ Alto(mE) $\Rightarrow_{\text{L}}$
    (π_1(Valor(mE,i,j)) $\iff$
    ($\exists n$: nombre)(def?(n,e.estudiantes) $\wedge_{\text{L}}$
    obtener(n,e.estudiantes) = $\langle i, j \rangle \wedge$ π_1(Siguiente(π_2(Valor(mE,i,j)))) = $n \wedge$
    esPermutacion(SecuSuby(π_2(Valor(mE,i,j)),secuenciarDic(e.estudiantes)))

13. #claves(e.agentes) ≥ 1 $\Rightarrow$ ($\exists p$: placa) def?(p,e.agentes) $\wedge_{\text{L}}$
    (obtener(p,e.agentes).pl = e.masVigilante.pl $\wedge$
    obtener(p,e.agentes).premios = e.masVigilante.premios $\wedge$
    (($\forall p'$: placa) def?(p',e.agentes) $\wedge p \neq p' \Rightarrow_{\text{L}}$
    (obtener(p,e.agentes).premios $\geq$ obtener(p'.e.agentes).premios $\wedge$
    (obtener(p,e.agentes).premios = obtener(p',e.agentes) $\Rightarrow$ $p < p'$))))

armarMultiPos : estr $\longrightarrow$ multiconj(posicion)
armarMultiPos(e) $\equiv$ multiSignificados(soloPosAgentes(e.agentes,claves(e.agentes)),claves(e.placasPos)) $\cup$
multiSignificados(e.hippies,claves(e.hippies)) $\cup$
multiSignificados(e.estudiantes,claves(e.estudiantes)

soloPosAgentes : dicc(nat $\times$ datosAg) d $\times$ conj(β) c $\longrightarrow$ dicc(nat,posicion) $\{c \subseteq claves(d)\}$
soloPosAgentes(d,c) $\equiv$ **if** $\emptyset?(c)$ **then**
Vacio()
else
definir(dameUno(c), obtener(dameUno(c),d).pos,soloPosAgentes(d,sinUno(c)))
fi

multiSignificados : dicc($\beta \times \alpha$) d $\times$ conj(β) c $\longrightarrow$ multiconj(α) $\{(\forall cl : \beta)\ cl \in c \Rightarrow \text{def?(cl,d)}\}$
multiSignificados(d,c) $\equiv$ **if** $\emptyset?(c)$ **then** $\emptyset$ **else** Ag(obtener(dameUno(c),d),multiSignificados(d,sinUno(c))) **fi**

unionSignificados : dicc($\beta \times \alpha$) d $\longrightarrow$ conj(α)
unionSignificados(d) $\equiv$ multiAConj(multiSignificados(d,claves(d)))

multiAConj : multiconj(α) $\longrightarrow$ conj(α)
multiAConj(mc) $\equiv$ **if** $mc = \emptyset$ **then** $\emptyset$ **else** Ag(dameUno(mc),multiAConj(sinUno(mc))) **fi**

secuenciarDic : dicc($\beta \times \alpha$) $\longrightarrow$ secu($\langle \beta, alpha \rangle$)
secuenciarDic(d) $\equiv$ secuenciarDicAux(d,claves(d)

secuenciarDicAux : dicc($\beta \times \alpha$) d $\times$ conj(β) c $\longrightarrow$ secu($\langle \beta, \alpha \rangle$) $\{(\forall cl : \beta)\ cl \in c \Rightarrow \text{def?(cl,d)}\}$
secuenciarDicAux(d,c) $\equiv$ **if** $\emptyset?(c)$ **then**
<>
else
$\langle$dameUno(c),obtener(dameUno(c),d)$\rangle$ $\bullet$ secuenciarDicAux(d,sinUno(c))
fi

ordenarπ_1 : secu($\langle \beta \times \alpha \rangle$) $\longrightarrow$ secu($\langle \beta, \alpha \rangle$)
ordenarπ_1(s) $\equiv$ **if** vacia?(s) **then** <> **else** minπ_1(s) $\bullet$ ordenarπ_1(sacar(minπ_1(s),s)) **fi**

minπ_1 : secu($\langle \beta \times \alpha \rangle$) s $\longrightarrow$ $\langle \beta, \alpha \rangle$ $\{\neg$vacia?(s)$\}$
minπ_1(s) $\equiv$ minAuxπ_1(s,prim(s))

minAuxπ_1 : secu($\langle \beta \times \alpha \rangle$) s $\times$ $\langle \beta \times \alpha \rangle$ $\longrightarrow$ $\langle \beta, \alpha \rangle$ $\{\neg$vacia?(s)$\}$
minAuxπ_1(s,e) $\equiv$ **if** long(s) $= 1$ **then**
if π_1(prim(s)) $<\pi_1(e)$ **then** prim(s) **else** e **fi**
else
minAuxπ_1(fin(s), **if** π_1(prim(s)) $<\pi_1(e)$ **then** prim(s) **else** e **fi**)
fi

sacar : α $\times$ secu(α) s $\longrightarrow$ secu(α) $\{$esta?(a,s)$\}$
sacar(a,s) $\equiv$ **if** prim(s) $= a$ **then** fin(s) **else** prim(s) $\bullet$ sacar(a,fin(s)) **fi**

Abs : estr e $\longrightarrow$ campusSeg $\{$Rep(e)$\}$
Abs(e) $\equiv$ cs: campusSeg / e.cp = campus(cs) $\wedge$ claves(e.estudiantes) = estudiantes(cs)
$\wedge$ claves(e.hippies) = hippies(cs) $\wedge$ claves(e.agentes) = agentes(cs) $\wedge$
($\forall a$: agente)(($a \in$ agentes(cs) $\iff$ def?(a,e.agentes)) $\wedge_{\text{L}}$ ($a \in$ agentes(cs) $\Rightarrow_{\text{L}}$
(obtener(a,e.agentes).sanciones = cantSanciones(a,cs) $\wedge$
obtener(a,e.agentes).premios = cantHippiesAtrapados(a,cs) $\wedge$
obtener(a,e.agentes).posicion = posAgente(a,cs)))) $\wedge_{\text{L}}$
($\forall n$: nombre)(def?(n,e.hippies) $\Rightarrow_{\text{L}}$ posEstudianteYHippie(n,cs) = obtener(n,e.hippies)) $\wedge$
(def?(n,e.estudiantes) $\Rightarrow_{\text{L}}$ posEstudianteYHippie(n,cs) = obtener(n,e.estudiantes))

Algoritmos

Algorithm 1 Campus

```
1: procedure iCampus(in c : estr) → res : campus
2:     res ← c.cp                                                    ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 2 Estudiantes

```
1: procedure iEstudiantes(in c : estr) → res : itClavesDiccN(string)
2:     res ← CrearIt(c.estudiantes)                                  ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 3 Hippies

```
1: procedure iHippies(in c : estr) → res : itClavesDiccN(string)
2:     res ← CrearIt(c.hippies)                                      ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 4 Agentes

```
1: procedure iAgentes(in c : estr) → res : itClavesDiccLog(nat)
2:     res ← CrearIt(c.agentes)                                      ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 5 Posicion de Estudiante Y Hippie

```
1: procedure iPosEstudianteYHippie(in c : estr, in n : nombre) → res : posicion
2:     if Definido(c.hippies,n then                                  ▷ O(|h_m|)
3:         res ← Obtener(n,c.hippies)                                ▷ O(|h_m|)
4:     else
5:         if Definido(c.estudiantes,n) then                         ▷ O(|e_m|)
6:             res ← Obtener(n,c.estudiantes)                        ▷ O(|e_m|)
```

Complejidad: $\mathcal{O}(|n_m|)$
Justificacion: h_m y e_m son el hippie y estudiante con nombre mas largo, sea cual sea la rama del if a la que se entra la complejidad llega a ser $\mathcal{O}(|n_m|) + \mathcal{O}(|n_m|) = \mathcal{O}(|n_m|)$ con $|n_m| = \max(|h_m|,|e_m|)$

Algorithm 6 Posicion de Agente

```
1: procedure iPosAgente(in c : estr, in p : placa) → res : posicion
2:     res ← Obtener(c.agentes, pl)                                  ▷ O(1) promedio
```

Complejidad: $\mathcal{O}(1)$ promedio

Algorithm 7 Cantidad de Sanciones

```
1: procedure iCantSanciones(in c : estr, in pl : placa) → res : nat
2:     res ← Obtener(c.agentes,pl).sanciones                         ▷ O(1) promedio
```

Complejidad: $\mathcal{O}(1)$ promedio

Algorithm 8 Cantidad de Hippies Atrapados

```
1: procedure iCantHippiesAtrapados(in c : estr, in pl : placa) → res : nat
2:     res ← Obtener(c.agentes,pl).premios                          ▷ O(1) promedio
```

Complejidad: $\mathcal{O}(1)$ promedio

Algorithm 9 Comenzar Rastrillaje

```
1: procedure iComenzarRastrillaje(in c : campus, in d : diccLineal(placa,posicion)) → res : estr
2:     res.cp ← Copiar(c)                                                          ▷ O(h * a)
3:     res.posiciones.mAg ← CrearMatriz(Filas(c),Columnas(c), CrearItDiccLog())    ▷ O(h * a)
4:     res.posiciones.mH ← CrearMatriz(Filas(c),Columnas(c), CrearItDiccN())       ▷ O(h * a)
5:     res.posiciones.mE ← CrearMatriz(Filas(c),Columnas(c), CrearItDiccN())       ▷ O(h * a)
6:     res.hippies ← Vacio()                                                       ▷ O(1)
7:     res.estudiantes ← Vacio()                                                   ▷ O(1)
8:     res.sanciones.lista ← Vacia()                                               ▷ O(1)
9:     res.sanciones.rapidas ← Vacio()                                             ▷ O(1)
10:    res.sanciones.actualizar ← true                                             ▷ O(1)
11:    res.agentesRapido ← Vacio(#Claves(d))                                       ▷ O(1)
12:    res.agentes ← Vacio(#Claves(d))                                             ▷ O(1)
13:    itDicc(placa,posicion) it ← CrearIt(d)                                      ▷ O(1)
14:    if HaySiguiente(it) then                                                    ▷ O(1)
15:        conj(placa) cp ← Vacio()                                                ▷ O(1)
16:        itLista(⟨nat,conj(placa)⟩) itlist ← AgregarAtras(res.sanciones.lista, cp)   ▷ (un conjunto vacio) O(1)
17:        bool f ← true                                                           ▷ O(1)
18:        while HaySiguiente(it) do                                               ▷ n * O(1)
19:            placa pl ← Copia(SiguienteClave(it))                                ▷ O(1)
20:            posicion pos ← CrearPosicion(X(SiguienteSignificado(it)),Y(SiguienteSignificado(it)))   ▷ O(1)
21:            itConj(placa) itc ← AgregarRapido(Siguiente(itlist).s, pl)          ▷ O(1)
22:            datosAg ag ← ⟨pl,0,0,pos,itlist,itc⟩                                ▷ O(1)
23:            itDicLog(nat,datosAg) itdl ← DefinirLento(res.agentes,pl,ag)        ▷ O(n)
24:            res.posiciones.mAg[X(pos)][Y(pos)] ← itdl                           ▷ O(1)
25:            Definir(res.agentesRapido,pl,itdl)                                  ▷ O(1) promedio
26:            if (f || res.masVigilante.placa ≥ pl) then                          ▷ O(1)
27:                res.masVigilante.pl ← ag.pl                                     ▷ O(1)
28:                res.masVigilante.premios ← ag.premios                           ▷ O(1)
29:                f ← false                                                       ▷ O(1)
```

Complejidad: $\mathcal{O}(h * a + n^2) + \mathcal{O}(n)$ prom
Donde

- h es Filas(c)
- a es Columnas(c)
- n es #Claves(d)

Justificacion: Sumo los $\mathcal{O}(1)$ continuos para aclarar un poco (C*$\mathcal{O}(1) = \mathcal{O}(1)$)
$4 * \mathcal{O}(h * a) + \mathcal{O}(1) + n * \mathcal{O}(1) * (\mathcal{O}(n) + \mathcal{O}(1)$ prom$) = \mathcal{O}(h * a) + \mathcal{O}(n) * (\mathcal{O}(n) + \mathcal{O}(1)$ prom $= \mathcal{O}(h * a) + \mathcal{O}(n^2) + \mathcal{O}(n)$ prom $= \mathcal{O}(h * a + n^2) + \mathcal{O}(n)$ prom

Algorithm 10 Ingresar Estudiante

```
1: procedure iIngresarEstudiante(in/out c : estr, in n : nombre, in p : posicion)
2:     c.pos.mE[X(p)][Y(p)] ← ⟨true, Definir(c.estudiantes,n,p)⟩          ▷ O(|e_m|)
3:     enhippizar(c, p)                                                  ▷ O(|h_m|)
4:     estudiantizar(c, p)                                               ▷ O(|e_m|)
5:     premiar(c,p)                                                      ▷ O(1)
6:     capturar(c,p)                                                     ▷ O(|h_m|)
7:     sancionar(c,p)                                                    ▷ O(1)
```

Complejidad: $\mathcal{O}(|n_m|)$
Justificacion: $\mathcal{O}(|e_m|) + \mathcal{O}(|h_m|) + \mathcal{O}(|e_m|) + \mathcal{O}(|h_m|) = \mathcal{O}(2|e_m|) + \mathcal{O}(|2h_m|) = \mathcal{O}(2max(|e_m|, |h_m|)) = \mathcal{O}(|n_m|)$

Algorithm 11 Ingresar Hippie

```
1: procedure iIngresarHippie(in/out c : estr, in n : nombre, in p : posicion)
2:     c.pos.mH[X(p)][Y(p)] ← ⟨true, Definir(c.hippies,n,p) ⟩            ▷ O(1)
3:     enhippizar(c, p)                                                  ▷ O(|h_m|)
4:     premiar(c,p)                                                      ▷ O(1)
5:     capturar(c,p)                                                     ▷ O(1)
6:     sancionar(c,p)                                                    ▷ O(1)
```

Complejidad: $\mathcal{O}(|n_m|)$
Justificacion: $\mathcal{O}(|h_m|) = \mathcal{O}(|n_m|)$

Algorithm 12 Mover Estudiante

```
1: procedure iMoverEstudiante(in/out c : estr, in n : nombre, in d : direccion)
2:     posicion p ← PosEstudianteYHippie(c,n)                            ▷ O(|n_m|)
3:     c.pos.mE[X(p)][Y(p)].def ← false                                  ▷ O(1)
4:     posicion proxp ← ProxPosicion(c.cp,p,d)                           ▷ O(1)
5:     if EsIngreso(c.cp,proxp) then                                     ▷ O(1)
6:         EliminarSiguiente(c.pos.mE[X(p)][Y(p)].dato)                  ▷ O(|e_m|)
7:     else
8:         c.pos.mE[X(proxp)][Y(proxp)].def ← true                       ▷ O(1)
9:         c.pos.mE[X(proxp)][Y(proxp)].dato ← c.pos.mE[X(p)][Y(p)].dato ▷ O(1)
10:        enhippizar(c, p)                                              ▷ O(|h_m|)
11:        estudiantizar(c, p)                                           ▷ O(|e_m|)
12:        premiar(c,p)                                                  ▷ O(1)
13:        capturar(c,p)                                                 ▷ O(|h_m|)
14:        sancionar(c,p)                                                ▷ O(1)
```

Complejidad: $\mathcal{O}(|n_m|)$
Justificacion: $\mathcal{O}(|n_m|) + \mathcal{O}(|h_m|) + \mathcal{O}(|e_m|) + \mathcal{O}(|h_m|) = \mathcal{O}(|n_m|) + \mathcal{O}(2|h_m|) + \mathcal{O}(|e_m|) = \mathcal{O}(|n_m|) + \mathcal{O}(|e_m| + |h_m|)$
$= \mathcal{O}(|n_m|) + \mathcal{O}(|n_m|) = \mathcal{O}(|n_m|)$

Algorithm 13 Mover Hippie

```
1:  procedure iMoverHippie(in/out c : estr, in n : nombre)
2:      posicion p ← PosEstudianteYHippie(c,n)                                  ▷ O(|n_m|)
3:      c.pos.mH[X(p)][Y(p)].def ← false                                        ▷ O(1)
4:      posicion proxp ← nuevaPosHippieOAgente(c,p,CrearIt(c.estudiantes))      ▷ O(N_e)
5:      c.pos.mH[X(proxp)][Y(proxp)].def ← true                                 ▷ O(1)
6:      c.pos.mH[X(proxp)][Y(proxp)].dato ← c.pos.mH[X(p)][Y(p)].dato           ▷ O(1)
7:      enhippizar(c, p)                                                        ▷ O(|h_m|)
8:      premiar(c,p)                                                            ▷ O(1)
9:      capturar(c,p)                                                           ▷ O(|h_m|)
10:     sancionar(c,p)                                                          ▷ O(1)
```

Complejidad: $\mathcal{O}(|n_m|) + \mathcal{O}(N_e)$
Justificacion: $\mathcal{O}(|n_m|) + \mathcal{O}(N_e) + \mathcal{O}(|h_m|) + \mathcal{O}(|h_m|) = \mathcal{O}(n_m) + \mathcal{O}(N_e) + 2\mathcal{O}(|h_m|) = \mathcal{O}(|n_m|) + \mathcal{O}(N_e) + 2\mathcal{O}(|n_m|)$
$= 3\mathcal{O}(|n_m|) + \mathcal{O}(N_e) = \mathcal{O}(|n_m|) + \mathcal{O}(N_e)$

Algorithm 14 Mover Agente

```
1:  procedure iMoverAgente(in/out c : estr, in pl : placa)
2:      posicion p ← Obtener(c.agentes,pl)                                      ▷ O(logN_a)
3:      c.pos.mAg[X(p)][Y(p)].def ← false                                       ▷ O(1)
4:      posicion proxp ← nuevaPosHippieOAgente(c,p,CrearIt(c.hippies))          ▷ O(N_h)
5:      c.pos.mAg[X(proxp)][Y(proxp)].def ← true                                ▷ O(1)
6:      c.pos.mAg[X(proxp)][Y(proxp)].dato ← c.pos.mAg[X(p)][Y(p)].dato         ▷ O(1)
7:      premiar(c,p)                                                            ▷ O(1)
8:      capturar(c,p)                                                           ▷ O(|h_m|)
9:      sancionar(c,p)                                                          ▷ O(1)
```

Complejidad: $\mathcal{O}(|n_m|) + \mathcal{O}(logN_a) + \mathcal{O}(N_h)$

Algorithm 15 Con K Sanciones

```
1:  procedure iConKSanciones(in/out c : estr, in k : nat) → res:conj(placa)
2:      if c.sanciones.actualizar then                                          ▷ O(1)
3:          actualizarSanciones(c)                                              ▷ O(N_a)
4:      if Def(c.sanciones,k) then                                              ▷ O(logN_a)
5:          res ← Obtener(c.sanciones.rapidas,k)                                ▷ O(logN_a)
6:      else
7:          res ← Vacio()                                                       ▷ O(1)
```

Complejidad: $\mathcal{O}(N_a)$ si hubo sanciones entre la ultima con K sanciones o $\mathcal{O}(logN_a)$ si no.

Algorithm 16 Con Mismas Sanciones

```
1:  procedure iConMismasSanciones(in c : estr, in pl : placa) → res:conj(placa)
2:      res ← Siguiente(Obtener(c.agentesRapido,pl).kSanc)                      ▷ O(1) promedio
```

Complejidad: $\mathcal{O}(1)$ promedio

Algorithm 17 Cantidad de Hippies

```
1:  procedure iCantHippies(in c : estr) → res:nat
2:      res ← #Claves(c.hippies)                                                ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 18 Cantidad de Estudiantes

```
1: procedure iCantEstudiantes(in c : estr) → res:nat
2:     res ← #Claves(c.estudiantes)                                  ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 19 Mas Vigilante

```
1: procedure iMasVigilante(in c : estr) → res:placa
2:     res ← c.masVigilante.pl                                       ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 20 Actualizar Sanciones

```
1: procedure iActualizarSanciones(in/out c : estr)
2:     it ← CrearIt(e.sanciones.lista)                               ▷ O(1)
3:     while HaySiguiente(it) do                                     ▷ n * O(1)
4:         Definir(e.sanciones.rapidas, Siguiente(it).c, Siguiente(it).s)   ▷ O(#(Claves(Siguiente(it).s)))
5:         Avanzar(it)                                               ▷ O(1)
6:     c.sanciones.actualizar ← false                                ▷ O(1)
```

Complejidad: $\mathcal{O}(n)$ Justificacion: Estableciendo n como la cantidad de sanciones diferentes. Si bien el definir de diccionario logaritmico funciona por copia, la sumatoria de todos la conjuntos de la lista de sanciones es menor o igual n (en peor caso n). Por lo tanto la complejidad es, en todos los casos $\mathcal{O}(2n) = \mathcal{O}(n)$

Algorithm 21 Nueva Posicion Hippie o Agente

```
1: procedure nuevaPosHippieOAgente(in c : estr, in p : posicion, in/out itObj : itDiccN(string,posicion)) → res : posicion
2:     nat min ← 0                                                   ▷ O(1)
3:     posicion objetivo                                             ▷ O(1)
4:     while HaySiguiente(itObj) do                                  ▷ n * O(1)
5:         if min == 0 || min ≥ DistanciaPosiciones(p,SiguienteSignificado(itObj)) then   ▷ O(1)
6:             min ← DistanciaPosiciones(p,SiguienteSignificado(itObj))   ▷ O(1)
7:             objetivo ← SiguienteSignificado(itObj)                ▷ O(1)
8:         Avanzar(itObj)                                            ▷ O(1)
9:     if min == 0 then                                              ▷ O(1)
10:        objetivo ← Siguiente(CrearIt(IngresosMasCercanos(c.cp,p)))   ▷ O(1)
11:    conj(posicion) vecinosLibres ← libres(c,Vecinos(c.cp,p))      ▷ O(k)
12:    itconj(posicion) itVL ← CrearIt(vecinosLibres)                ▷ O(1)
13:    min ← 0                                                       ▷ O(1)
14:    while HaySiguiente(itVL) do                                   ▷ m * O(1)
15:        if min == 0 || min ≥ DistanciaPosiciones(objetivo,Siguiente(itVL)) then
16:            min ← DistanciaPosiciones(objetivo,Siguiente(itVL))   ▷ O(1)
17:            res ← Siguiente(itVL)                                 ▷ O(1)
18:        Avanzar(itVL)                                             ▷ O(1)
```

Complejidad: $\mathcal{O}(n)$ siendo n la cantidad de iteraciones de *itObj*
Justificacion: $n * Ogr(1) + \mathcal{O}(k) + \mathcal{O}(m) = \mathcal{O}(n + k + m)$
Donde

- n es la cantidad de iteraciones de *itObj*
- m es la cantidad de posiciones libres de vecinos de una posicion, acotado por 4
- k es la cantidad de vecinos de una posicion, acotado por 4

$\mathcal{O}(n + 4 + 4) = \mathcal{O}(n + 8) = \mathcal{O}(n)$

Algorithm 22 Posiciones Libres

```
1: procedure LIBRES(in c : estr, in cp : conj(posicion)) → res : conj(posicion)
2:     itconj(posicion) itc ← CrearIt(cp)                                  ▷ O(1)
3:     res ← Vacio()                                                       ▷ O(1)
4:     while HaySiguiente(itc) do                                          ▷ n * O(1)
5:         if !ocupada(c,Siguiente(p)) then                                ▷ O(1)
6:             AgregarRapido(res,Siguiente(p))                             ▷ O(1)
7:         Avanzar(itc)                                                    ▷ O(1)
```

Complejidad: $\mathcal{O}(n)$ siendo n #Claves(cp)

Algorithm 23 De estudiante a hippie

```
1: procedure ENHIPPIZAR(in/out c : estr, in p : posicion)
2:     itConj(posicion) itc ← CrearIt(AgregarRapido(p,Vecinos(c.cp,p)))    ▷ O(1)
3:     while HaySiguiente(itc) do                                          ▷ n + O(1)
4:         posicion p ← Siguiente(itc)                                     ▷ O(1)
5:         if esEstudiante(c,p) && enhippizado(c,p) then                   ▷ O(1)
6:             itDiccN(string,posicion) itEaH ← c.pos.mE[X(p)][Y(p)].dato  ▷ O(1)
7:             EliminarSiguiente(itEaH)                                    ▷ O(h_m)
8:             c.pos.mE[X(p)][Y(p)].def ← false                            ▷ O(1)
9:             c.pos.mH[X(p)][Y(p)].def ← true                             ▷ O(1)
10:            c.pos.mH[X(p)][Y(p)].dato ← Definir(c.hippies,n,p)          ▷ O(|h_m|)
11:        Avanzar(itc)                                                    ▷ O(1)
```

Complejidad: $\mathcal{O}(|h_m|)$
Justificacion: $\mathcal{O}(1) + (n * \mathcal{O}(1)) * 2\mathcal{O}(|h_m|) = \mathcal{O}(n) * 2\mathcal{O}(|h_m|)$ con n siendo la cantidad de vecinos + 1, acotado por 5. $\mathcal{O}(5) * 2\mathcal{O}(|h_m|) = \mathcal{O}(|h_m|)$

Algorithm 24 De hippie a estudiante

```
1: procedure ESTUDIANTIZAR(in/out c : estr, in p : posicion)
2:     itConj(posicion) itc ← CrearIt(AgregarRapido(p,Vecinos(c.cp,p)))    ▷ O(1)
3:     while HaySiguiente(itc) do                                          ▷ n + O(1)
4:         posicion p ← Siguiente(itc)                                     ▷ O(1)
5:         if esHippie(c,p) && estudiantizado(c,p) then                    ▷ O(1)
6:             itDiccN(string,posicion) itHaE ← c.pos.mH[X(p)][Y(p)]       ▷ O(1)
7:             EliminarSiguiente(itHaE)                                    ▷ O(|e_m|)
8:             c.pos.mH[X(p)][Y(p)].def ← false                            ▷ O(1)
9:             c.pos.mE[X(p)][Y(p)].def ← true                             ▷ O(1)
10:            c.pos.mE[X(p)][Y(p)].dato ← Definir(c.estudiantes,n,p)      ▷ O(|e_m|)
```

Complejidad: $\mathcal{O}(|e_m|)$
Justificacion: $\mathcal{O}(1) + (n * \mathcal{O}(1)) * 2\mathcal{O}(|e_m|) = \mathcal{O}(n) * 2\mathcal{O}(|e_m|)$ con n siendo la cantidad de vecinos + 1, acotado por 5. $\mathcal{O}(5) * 2\mathcal{O}(|e_m|) = \mathcal{O}(|e_m|)$

Algorithm 25 Premiar Agentes

```
1:  procedure PREMIAR(in/out c : estr, in p : posicion)
2:      if rodeado(c,p) && esHippie(c,p) && #Claves(agentes(c,Vecinos(c.cp,p)) then        ▷ O(k)
3:          itConj(posicion) itca ← agentes(c,Vecinos(c.cp,p))                            ▷ O(k)
4:          while HaySiguiente(itca) do                                                   ▷ n * O(1)
5:              Siguiente(c.pos.mAg[X(p)][Y(p)]).premios ++                               ▷ O(1)
6:              actualizarMasVigilante(c,pos.mAg[X(p)][Y(p)])                             ▷ O(1)
7:              Avanzar(itca)                                                             ▷ O(1)
8:      else
9:          itConj(posicion) itc ← CrearIt(Vecinos(c.cp, p))                              ▷ O(1)
10:         while HaySiguiente(itc) do                                                    ▷ n * O(1)
11:             posicion p ← Siguiente(itc)                                               ▷ O(1)
12:             if rodeado(p) && esHippie(p) then                                         ▷ O(1)
13:                 itConj(posicion) itca ← agentes(c,Vecinos(c.cp,p))                    ▷ O(k)
14:                 while HaySiguiente(itca) do                                           ▷ m * O(1)
15:                     pa ← Siguiente(itca)                                              ▷ O(1)
16:                     Siguiente(pos.mAg[X(pa)][Y(pa)]).premios ++                       ▷ O(1)
17:                     actualizarMasVigilante(c,pos.mAg[X(pa)][Y(pa)])                   ▷ O(1)
18:                     Avanzar(itca)                                                     ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificacion: $\mathcal{O}(1) + n * \mathcal{O}(1) * (\mathcal{O}(k) + m\mathcal{O}(1)) = \mathcal{O}(n * (k + m))$
Donde

- n es la cantidad de vecinos de una posicion, acotado por 4
- m es la cantidad de agentes vecinos de una posicion, acotado por 4
- k es la cantidad de vecinos de una posicion, acotado por 4

$\mathcal{O}(4 * (4 + 4)) = \mathcal{O}(32) = \mathcal{O}(1)$

Algorithm 26 Capturar Hippies

```
1:  procedure CAPTURAR(in/out c : estr, in p : posicion)
2:      if rodeado(c,p) && esHippie(c,p) && #Claves(agentes(c,Vecinos(c.cp,p)) ≥ 1 then   ▷ O(k)
3:          itDiccN(string,posicion) itH ← res.pos.mH[X(p)][Y(p)]                         ▷ O(1)
4:          EliminarSiguiente(itH)                                                        ▷ O(|h_m|)
5:          res.pos.mE[X(p)][Y(p)].def ← false                                            ▷ O(1)
6:      else
7:          itConj(posicion) itc ← CrearIt(Vecinos(c.cp,p))                               ▷ O(1)
8:          while HaySiguiente(itc) do                                                    ▷ n * O(1)
9:              posicion p ← Siguiente(itc)                                               ▷ O(1)
10:             if rodeado(c,p) && esHippie(c,p) && #Claves(agentes(c,Vecinos(c.cp,p))) ≥ 1 then   ▷ O(k)
11:                 itDiccN(string,posicion) itH ← c.pos.mH[X(p)][Y(p)]                   ▷ O(1)
12:                 EliminarSiguiente(itH)                                                ▷ O(|h_m|)
13:                 c.pos.mE[X(p)][Y(p)].def ← false                                      ▷ O(1)
14:             Avanzar(itc)                                                              ▷ O(1)
```

Complejidad: $\mathcal{O}(|h_m|)$
Justificacion: $\mathcal{O}(k) + n * \mathcal{O}(1) * (\mathcal{O}(|h_m|) + \mathcal{O}(k)) = \mathcal{O}(k) + \mathcal{O}(n) * \mathcal{O}(|h_m| + k) = \mathcal{O}(k) + \mathcal{O}(n * |h_m| + n * k) = \mathcal{O}(n * |h_m| + n * k)$ $(n * k > k)$
Donde

- n es la cantidad de vecinos de una posicion, acotado por 4
- $|h_m|$ es el hippie con nombre mas largo
- k es la cantidad de vecinos de una posicion, acotado por 4

$\mathcal{O}(4 * |h_m| + 4 * 4) = \mathcal{O}(4 * |h_m| + 16) = \mathcal{O}(4 * |h_m|) = \mathcal{O}(|h_m|)$

Algorithm 27 Sancionar Agentes

```
1:  procedure SANCIONAR(in/out c : estr, in p : posicion)
2:      itConj(posicion) itc ← CrearIt(AgregarRapido(p,Vecinos(c.cp,p))          ▷ O(1)
3:      while HaySiguiente(itc) do                                               ▷ n * O(1)
4:          posicion p ← Siguiente(itc)                                          ▷ O(1)
5:          conj(posicion) cag ← agentes(c,Vecinos(c.cp,p)                       ▷ O(k)
6:          if rodeado(c,p) && esEstudiante(c,p) && #Claves(cag) ≥ 1 then        ▷ O(1)
7:              itConj(posicion) itca ← cag                                      ▷ O(1)
8:              while HaySiguiente(itca) do                                      ▷ m * O(1)
9:                  c.sanciones.actualizar ← true                                ▷ O(1)
10:                 posicion pa ← Siguiente(itca)                                ▷ O(1)
11:                 datosAg datos ← Siguiente(c.pos.mAg[X(pa)][Y(pa)]).dato      ▷ O(1)
12:                 datos.sanciones ++
13:                 EliminarSiguiente(datos.mismasSanc)                          ▷ O(1)
14:                 if #(Siguiente(datos.kSanc).s) == 0 then                     ▷ O(1)
15:                     EliminarSiguiente(datos.kSanc)                           ▷ O(1)
16:                 if datos.sanciones < SiguienteClave(datos.kSanc) then        ▷ O(1)
17:                     InsertarAdelante(datos.kSanc,datos.sanciones,Vacio())    ▷ O(1)
18:                 else
19:                     Avanzar(datos.kSanc)                                     ▷ O(1)
20:                 datos.mismasSanc ← AgregarRapido(SiguienteSignificado(datos.kSanc),datos.pl)  ▷ O(1)
21:         Avanzar(itc)                                                         ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificacion: $n * (\mathcal{O}(k) + m * \mathcal{O}(1)) = \mathcal{O}(n) * \mathcal{O}(k + m) = \mathcal{O}(n * k + n * m)$
Donde

- n es la cantidad de vecinos de una posicion + 1, acotado por 5
- m es la cantidad de agentes vecinos de una posicion, acotado por 4
- k es la cantidad de vecinos de una posicion, acotado por 4

$\mathcal{O}(5 * 4 + 5 * 4) = \mathcal{O}(40) = \mathcal{O}(1)$

Algorithm 28 ¿Se transforma en hippie?

```
1:  procedure ENHIPPIZADO(in c : estr, in p : posicion) → res : bool
2:      res ← false                                                              ▷ O(1)
3:      itConj(posicion) itc ← CrearIt(Vecinos(c.cp, p))                         ▷ O(1)
4:      nat i ← 0                                                                ▷ O(1)
5:      while HaySiguiente(itc) do                                               ▷ n * O(1)
6:          if esHippy(c,Siguiente(itc)) then                                    ▷ O(1)
7:              i ← i + 1                                                        ▷ O(1)
8:          Avanzar(itc)                                                         ▷ O(1)
        res ← (i ≥ 2)                                                            ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificacion: $\mathcal{O}(1) + n * \mathcal{O}(1) = \mathcal{O}(n)$ siendo n la cantidad de vecinos, que como maximo es 4
$\mathcal{O}(4) = \mathcal{O}(1)$

Algorithm 29 ¿Es hippie?

```
1:  procedure ESESTUDIANTE(in c : estr, in p : posicion) → res : bool
2:      res ← c.pos.mH[X(p)][Y(p)].def                                           ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 30 ¿Se transforma en estudiante?

```
1: procedure ENHIPPIZADO(in c : estr, in p : posicion) → res : bool
2:     res ← false                                                    ▷ O(1)
3:     itConj(posicion) itc ← CrearIt(Vecinos(c.cp, p))               ▷ O(1)
4:     nat i ← 0                                                      ▷ O(1)
5:     while HaySiguiente(itc) do                                     ▷ n * O(1)
6:         if esEstudiante(Siguiente(itc)) then                       ▷ O(1)
7:             i ← i + 1                                              ▷ O(1)
8:         Avanzar(itc)                                               ▷ O(1)
       res ← (i = 4)                                                  ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificacion: $\mathcal{O}(1) + n * \mathcal{O}(1) = \mathcal{O}(n)$ siendo n la cantidad de vecinos, que como maximo es 4
$\mathcal{O}(4) = \mathcal{O}(1)$

Algorithm 31 ¿Es estudiante?

```
1: procedure ESESTUDIANTE(in c : estr, in p : posicion) → res : bool
2:     res ← c.pos.mE[X(p)][Y(p)].def                                 ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 32 ¿Esta Rodeado?

```
1: procedure RODEADO(in c : estr, in p : posicion) → res : bool
2:     itConj(posicion) itc ← CrearIt(Vecinos(c.cp, p))               ▷ O(1)
3:     nat i ← 0                                                      ▷ O(1)
4:     while HaySiguiente(itc) do                                     ▷ n * O(1)
5:         if ocupado(Siguiente(p)) then                              ▷ O(1)
6:             i++                                                    ▷ O(1)
           Avanzar(itc)                                               ▷ O(1)
       res ← (i == 4)                                                 ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$
Justificacion: $n * \mathcal{O}(1) = \mathcal{O}(n)$ siendo n la cantidad de vecinos de una posicion, con cota 4
$\mathcal{O}(4) = \mathcal{O}(1)$

Algorithm 33 ¿Esta Ocupado?

```
1: procedure RODEADO(in c : estr, in p : posicion) → res : bool
2:     res ← (esHippie(c,p) || esAgente(c,p) || esEstudiante(c,p) || Ocupada(c.cp,p))   ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 34 Agentes en posiciones

```
1: procedure AGENTES(in c : estr, in cp : conj(posicion)) → res : conj(posicion)
2:     itConj(posicion) itc ← CrearIt(cp)          ▷ O(1) res ←Vacio();
3:     while HaySiguiente(itc) do                  ▷ n * O(1)
4:         if esAgente(Siguiente(p)) then          ▷ O(1)
5:             AgregarRapido(res,p)                ▷ O(1)
           Avanzar(itc)                            ▷ O(1)
```

Complejidad: $\mathcal{O}(n)$ siendo $n = \#(\text{Claves}(cp))$

Algorithm 35 ¿Es agente?

```
1: procedure esAgente(in c : estr, in p : posicion) → res : bool
2:     res ← c.pos.mAg[X(p)][Y(p)].def                                  ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 36 Actualizar mas Vigilante

```
1: procedure actualizarVigilante(in/out c : estr, in datos : datosAg)
2:     if datos.premios > c.masVigilante.premios then                   ▷ O(1)
3:         c.masVigilante.pl ← datos.pl                                 ▷ O(1)
4:         c.masVigilante.premios ← datos.premios                       ▷ O(1)
5:     if datos.premios == c.masVigilante.premios then                  ▷ O(1)
6:         if datos.pl < c.masVigilante.pl then                         ▷ O(1)
7:             c.masVigilante.pl ← datos.pl                             ▷ O(1)
8:             c.masVigilante.premios ← datos.premios                   ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

2.2. Campus

Interfaz

se explica con: CAMPUS

usa: nat, bool, Matriz(bool), posicion

géneros: `campus`

Operaciones de Campus

CREARCAMPUS(**in** *ancho*: `nat`, **in** *alto*: `nat`) → *res* : `campus`
Pre ≡ {ancho > 0 ∧ alto > 0}
Post ≡ {res =$_{obs}$ crearCampus(ancho,alto)}
Complejidad: $\mathcal{O}$(alto x ancho)
Descripción: Crea un campus sin obstáculos.

AGREGAROBSTACULO(**in/out** c: `campus`, **in** p: `posicion`)
Pre ≡ {posValida?(p,c) ∧ ¬ocupada(p, c) ∧ c = c_0}
Post ≡ {c =$_{obs}$ agregarObstáculo(p, c_0)}
Complejidad: $\mathcal{O}$(1)
Descripción: Agrega un obstáculo en la posición dada.

FILAS(**in** c: `campus`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ filas(c)}
Complejidad: $\mathcal{O}$(1)
Descripción: Indica la cantidad de filas del campus dado.

COLUMNAS(**in** c: `campus`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ columnas(c)}
Complejidad: $\mathcal{O}$(1)
Descripción: Indica la cantidad de columnas del campus dado.

OCUPADA(**in** c: `campus`, **in** p: `posicion`) → *res* : `bool`
Pre ≡ {posValida?(p,c)}
Post ≡ {res =$_{obs}$ ocupada(p,c)}
Complejidad: $\mathcal{O}$(1)
Descripción: Indica si la posición dada tiene un obstáculo.

POSVALIDA(**in** c: `campus`, **in** p: `posicion`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ posValida?(p,c)}
Complejidad: $\mathcal{O}$(1)

INGRESOSUPERIOR(**in** c: `campus`, **in** p: `posicion`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ ingresoSuperior?(p,c)}
Complejidad: $\mathcal{O}$(1)

INGRESOINFERIOR(**in** c: `campus`, **in** p: `posicion`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ ingresoInferior?(p,c)}
Complejidad: $\mathcal{O}$(1)

ESINGRESO(**in** c: `campus`, **in** p: `posicion`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ esIngreso?(p,c)}

Complejidad: $\mathcal{O}(1)$

VECINOS(**in** c: `campus`, **in** p: `posicion`) $\rightarrow$ res : `conj(posicion)`
Pre $\equiv$ {posValida?(p,c)}
Post $\equiv$ {res $=_{obs}$ vecinos?(p,c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Dada una posición, devuelve las 4 posiciones vecinas.

VECINOSCOMUNES(**in** c: `campus`, **in** p_1 : `posicion`, **in** p_2 : `posicion`) $\rightarrow$ res : `conj(posicion)`
Pre $\equiv$ {posValida?(p_1,c) $\wedge$ posValida?(p_1,c)}
Post $\equiv$ {res $=_{obs}$ vecinosComunes(p_1,p_2,c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Dadas dos posiciones, devuelve las posiciones que son vecinas de ambas.

VECINOSVALIDOS(**in** c: `campus`, **in/out** cp: `conj(posicion)`)
Pre $\equiv$ {c = c_0}
Post $\equiv$ {c $=_{obs}$ vecinosValidos}
Complejidad: $\mathcal{O}(|cp|)$
Descripción: Dado un conjunto de posiciones, devuelve el conjunto resultado de quitar las posiciones no válidas.

PROXPOSICION(**in** c: `campus`, **in** p: `posicion`, **in** d: `direccion`) $\rightarrow$ res : `posicion`
Pre $\equiv$ {posValida?(p,c)}
Post $\equiv$ {res $=_{obs}$ proxPosición(p,d,c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el resultado de mover la posición indicada en la dirección dada.

INGRESOSMASCERCANOS(**in** c: `campus`, **in** p: `posicion`) $\rightarrow$ res : `conj(posicion)`
Pre $\equiv$ {posValida?(p,c)}
Post $\equiv$ {res $=_{obs}$ ingresosMásCercanos(p,c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto de ingresos más cercanos a la posición dada.

COPIAR(**in** c: `campus`) $\rightarrow$ res : `campus`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ c}
Complejidad: $\mathcal{O}$(Filas(c) * Columnas(c))
Descripción: Crea una copia del campus.

Representación

Representación del campus

`campus` se representa con `estr`

donde estr es Matriz(bool)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$ true

Abs : estr e $\longrightarrow$ campus {Rep(e)}
Abs(e) $\equiv$ c : campus /
filas(c) = Alto(e) $\wedge$ columnas(c) = Ancho(e)
$\wedge$ ($\forall$ p : posicion)(posValida?(p,c) $\Rightarrow_{\text{L}}$ ocupada?(p,c) = e[X(p) $-$ 1][Y(p) $-$ 1])

Algoritmos

Algorithm 37 iCrearCampus

1: **procedure** iCrearCampus(in ancho : nat, in alto : nat) → *res* : *estr*
2: *res* ← Crear(ancho,alto,false) ▷ $\mathcal{O}$(ancho x alto x copy(bool))

Complejidad: $\mathcal{O}$(ancho * alto) Justificación: $\mathcal{O}(\text{copy(bool)}) = \mathcal{O}(1)$

Algorithm 38 iAgregar

1: **procedure** iAgregar(in/out e : estr, in p : posicion)
2: e[X(p) − 1][Y(p) − 1] ← true ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 39 iFila

1: **procedure** iFila(in e : estr) → *res* : *nat*
2: *res* ← Alto(*e*) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 40 iColumna

1: **procedure** iColumna(in e : estr) → *res* : *nat*
2: *res* ← Ancho(*e*) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 41 iOcupada

1: **procedure** iOcupada(in e : estr, in p: posicion) → *res* : *bool*
2: *res* ← e[X(p) − 1][Y(p) − 1] ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 42 iPosValida

procedure iPosValida(in e: estr, in p : posicion) → *res* : *bool*
$0 < \text{X(p)} \land \text{X(p)} \leq \text{Ancho(e)} \land 0 < \text{Y(p)} \land \text{Y(p)} \leq \text{Alto(e)}$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 43 iIngresoSuperior

1: **procedure** iIngresoSuperior(in e : estr, in p : posicion) → *res* : *bool*
2: *res* ← (Y(p) = 1) ∧ iPosValida(e,p) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 44 iIngresoInferior

1: **procedure** iIngresoInferior(in e : estr, in p : posicion) → *res* : *bool*
2: *res* ← (Y(p) = Alto(e)) ∧ iPosValida(e,p) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 45 iEsIngreso

```
1: procedure iEsIngreso(in e : estr, in p : posicion) → res : bool
2:     res ← (iEsIngresoSuperior(e,p) ∨ iEsIngresoInferior(e,p))        ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 46 iVecinos

```
1: procedure iVecinos(in e : estr, in p : posicion) → res : conj(posicion)
2:     res ← Vacio() //vacio de conjunto                                ▷ O(1)
3:     AgregarRapido(res, <X(p) − 1, Y(p)>)                             ▷ O(1)
4:     AgregarRapido(res, <X(p) + 1, Y(p)>)                             ▷ O(1)
5:     AgregarRapido(res, <X(p), Y(p) − 1>)                             ▷ O(1)
6:     AgregarRapido(res, <X(p), Y(p) + 1>)                             ▷ O(1)
7:     res ← iVecinosValidos(e, res)                                    ▷ O(#res) = O(4) = O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 47 iVecinosComunes

```
1: procedure iVecinosComunes(in e : estr, in p_1 : posicion, in p_2 : posicion) → res : conj(posicion)
2:     vec_1 ← iVecinos(e, p_1)                                         ▷ O(1)
3:     vec_2 ← iVecinos(e, p_2)                                         ▷ O(1)
4:     res ← Vacio() //vacio de conjunto                                ▷ O(1)
5:     it_1 ← CrearIt(vec_1)                                            ▷ O(1)
6:     while HaySiguiente(it_1) do   ▷ Se repite 4 veces y lo de adentro es O(1) ⇒ O(1)
7:         it_2 ← CrearIt(vec_2)                                        ▷ O(1)
8:         while HaySiguiente(it_2) do   ▷ Se repite 4 veces y lo de adentro es O(1) ⇒ O(1)
9:             if Siguiente(it_1) = Siguiente(it_2) then                ▷ O(1)
10:                AgregarRapido(res, Siguiente(it_1))                  ▷ O(1)
11:            Avanzar(it_2)                                            ▷ O(1)
12:        Avanzar(it_1)                                                ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 48 iVecinosValidos

```
1: procedure iVecinosValidos(in e : estr,in/out cp : conj(posicion))
2:     it ← CrearIt(cp)                                                 ▷ O(1)
3:     while HaySiguiente(it) do   ▷ Se repite #cp veces y lo de adentro es O(1) ⇒ O(#cp)
4:         if ¬ iPosValida(e, Siguiente(it)) then                       ▷ O(1)
5:             EliminarSiguiente(it)                                    ▷ O(1)
6:         else
7:             Avanzar(it)                                              ▷ O(1)
```

Complejidad: $\mathcal{O}(\#cp)$

Algorithm 49 iProxPosicion

```
1: procedure iProxPosicion(in e : estr, in p : posicion, in dir : direccion) → res : posicion
2:     res ← <X(p) + β(dir = der) − β(dir = izq), Y(p) + β(dir = abajo) − β(dir = arriba)>   ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 50 iIngresosMasCercanos

```
1: procedure iIngresosMasCercanos(in e : estr, in p : posicion) → res : conj(posicion)
2:     res ← Vacio() //vacio de conjunto                                   ▷ O(1)
3:     if Dist(p, <X(p), 1>) < Dist(p, <Y(p), iFilas(e)>) then             ▷ O(1)
4:         AgregarRapido(res, <X(p), 1>)                                   ▷ O(1)
5:     else
6:         if Dist(p, <X(p), 1>) > Dist(p, <Y(p), iFilas(e)>) then         ▷ O(1)
7:             AgregarRapido(res, <X(p), iFilas(e)>)                       ▷ O(1)
8:         else
9:             AgregarRapido(res, <X(p), 1>)                               ▷ O(1)
10:            AgregarRapido(res, <X(p), iFilas(e)>)                       ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 51 iCopiar

```
1: procedure iCopiar(in e : estr) → res : estr
2:     res ← Copiar(e)                                                     ▷ O(Copiar(e))
```

Complejidad: $\mathcal{O}$(Ancho(e) * Alto(e))
Justificación: e src Matriz(bool)
⇒ Copiar(e) usa el copiar de matriz
⇒ $\mathcal{O}$(Copiar(e)) = $\mathcal{O}$(Ancho(e) * Alto(e) * copy(bool)) Además $\mathcal{O}$(bool) = $\mathcal{O}$(1)
⇒ $\mathcal{O}$(Copiar(e)) = $\mathcal{O}$(Ancho(e) * Alto(e))

2.3. Diccionario de Nombres

Interfaz

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

se explica con: DICCIONARIO(STRING,σ)

usa: nat, bool

géneros: `diccNom(string, σ)`

Operaciones de DiccionarioNombres(string,σ)

VACIO() $\rightarrow$ res : `diccNom(string, σ)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ vacio}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un diccionario vacio.

DEFINIR(**in/out** d: `diccNom(string,σ)`, **in** c: `string`, **in** s: σ) $\rightarrow$ res : `itDiccNom(string,σ)`
Pre $\equiv$ {$\neg$def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d $=_{obs}$ Definir(c,s,d_0) $\wedge$ haySiguiente(res) $\wedge_L$ siguiente(res) $=_{obs}$ <c,s>$\wedge$ alias(esPermutacion(SecuSuby(res),d))}
Complejidad: $\mathcal{O}$(|c| + copy(s))
Descripción: Agrega un elemento al diccionario y devuelve un iterador apuntando a ese elemento.

DEFINIDO(**in** d: `diccNom(string,σ)`, **in** c: `string`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ def?(c,d)}
Complejidad: $\mathcal{O}$(|c|)
Descripción: Devuelve verdadero si y sólo si la clave de entrada esta definida en el diccionario.

BORRAR(**in/out** d: `diccNom(string,σ)`, **in** c: `string`)
Pre $\equiv$ {def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d $=_{obs}$ borrar(c,d_0)}
Complejidad: $\mathcal{O}$(#claves(d) + |c|)
Descripción: Borra el elemento correspondiente a la clave c del diccionario.

BORRARRAPIDO(**in/out** d: `diccNom(string,σ)`, **in** it: `itDiccNom(string,σ)`)
Pre $\equiv$ {def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d $=_{obs}$ borrar(c,d_0)}
Complejidad: $\mathcal{O}$(|c|)
Descripción: Borra el elemento correspondiente a la clave c del diccionario. c es la clave correspondiente al siguiente del iterador.

OBTENER(**in/out** d: `diccNom(string,σ)`, **in** c: `string`) $\rightarrow$ res : σ
Pre $\equiv$ {def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {alias(res $=_{obs}$ obtener(c,d)) $\wedge$ ($\forall$ c' : string)(c' $\in$ claves(s) $\Rightarrow_L$ obtener(c,d) $=_{obs}$ obtener(c,d_0))}
Complejidad: $\mathcal{O}$(|c|)
Descripción: Devuelve el significado de la clave dada en el diccionario
Aliasing: res es modificable si y solo si σ es modificable

#CLAVES(**in** d: `diccNom(string,σ)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}

Post $\equiv$ {res $=_{obs}$ #claves(d)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de claves definidas en el diccionario

Representación

Representacion del DiccionaroNombres(string, σ)

`diccNom(string,`σ`)` **se representa con** `estr`

donde `estr` es `tupla`(*lineal*: `dicc(string,`σ`)` , *trie*: `diccT(string,`σ`)`)
Donde diccT(string,σ) es Lista(tupla<*letra* : char, *significado* : puntero(σ), *prox* : diccT(string, σ)>)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ Rep(e.lineal) $\wedge$ tamaño(e.trie) $<$ 256 $\wedge$ SinRepetidos(Letras(e.trie))
(∀ dT : diccT(string,σ)) (Esta?(dT, e.trie.prox) $\Rightarrow_{L}$ Rep(dT))
Letra : li $\longrightarrow$ secu(char)
Letra(l) $\equiv$ **if** vacio?(l) **then** <> **else** prim(l).letra • Letra(fin(l)) **fi**
Donde li es Lista(tupla<*letra* : char, *significado* : puntero(σ), *prox* : diccT(string, σ)>
SinRepetidos : secu(α) $\longrightarrow$ bool
SinRepetidos(s) $\equiv$ **if** vacio?(s) **then** true **else** ¬esta?(prim(s),fin(s)) $\wedge$ SinRepetidos(fin(s)) **fi**

Abs : estr e $\longrightarrow$ diccNom(string, σ) {Rep(e)}
Abs(e) $\equiv$ dN : diccNom(string, σ) /
Abs(e.lineal) $=_{obs}$ dN
$\wedge$ (∀ c : string) (Def?(c,d) $\Leftrightarrow$ Definido(c,e.trie))
$\wedge$ (∀ c : string) (Def?(c,d) $\Rightarrow_{L}$ Obtener(c,d) $=_{obs}$ Significado(c, e.trie))
Definido : string $\times$ trie $\longrightarrow$ bool
Definido(s,t) $\equiv$ estaLetra(s[0],t) $\wedge_{L}$ **if** Tamaño(s) == 1 **then** true **else** Definido(c[1:Tamaño(c)-1],t) **fi**

estaLetra : char $\times$ trie $\longrightarrow$ bool
estaLetra(c,t) $\equiv$ **if** vacio?(t) **then** false **else** Primero(t).letra $=_{obs}$ c $\vee$ estaLetra(c, fin(t)) **fi**

Significado : string s $\times$ trie t $\longrightarrow$ σ {Definido(s,t)}
Significado(s,e) $\equiv$ **if** Tamaño(s) = 1 **then**
*(Buscar(s[0],t).significado)
else
Significado(s[1:Tamaño(s)-1],Buscar(s[0],t).prox)
fi
Buscar : char c $\times$ trie t $\longrightarrow$ piso {estaLetra(c,t)}
Buscar(c,t) $\equiv$ **if** primero(t).letra = c **then** Primero(t) **else** Buscar(c,fin(t)) **fi**
Donde piso es tupla<*letra* : char, *significado* : puntero(σ), *prox* : trie>

Donde trie es diccT(string, σ)

Algoritmos

Algorithm 52 iVacio

procedure IVACIO $\rightarrow$ *res* : estr
res.lineal $\leftarrow$ Vacio() //vacio de diccionario lineal ▷ $\mathcal{O}(1)$
res.trie $\leftarrow$ Vacio() //vacio de lista ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 53 iDefinir

procedure IDEFINIR(**in/out** e : **estr**, **in** c : **string**, **in** s : σ) $\rightarrow$ *res* : *itDiccNom*(*string*, σ)
- *res* $\leftarrow$ DefinirRapido(e.lineal,c,s) ▷ $\mathcal{O}$(copy(s))
- *piso* $\leftarrow$ e.trie ▷ $\mathcal{O}(1)$
- *i* $\leftarrow$ 0
- **while** i < |c|-1 **do** ▷ Se repite |c|-1 veces y lo de adentro es $\mathcal{O}(1) \Rightarrow \mathcal{O}(|c|\text{-}1) = \mathcal{O}(|c|)$
  - **if** iEsta(c[i],piso) **then**
    - *piso* $\leftarrow$ iBuscar(c[i], piso).prox ▷ $\mathcal{O}(1)$
  - **else**
    - AgregarAdelante(piso, <c[i],NULL,Vacio()>) ▷ $\mathcal{O}(1)$
    - *piso* $\leftarrow$ piso[0].prox ▷ $\mathcal{O}(1)$
  - i++ ▷ $\mathcal{O}(1)$
- **if** iEsta(c[i],piso) **then** ▷ $\mathcal{O}(1)$
  - iBuscar(c[i],piso).significado $\leftarrow$ puntero(SiguienteSignificado(res)) ▷ $\mathcal{O}(1)$
- **else**
  - AgregarAdelante(piso, <c[i],puntero(SiguienteSignificado(res)),Vacio()>) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(|c| + \text{copy(s)})$

Algorithm 54 iDefinido

procedure IDEFINIDO(**in** e : **estr**, **in** c : **string**) $\rightarrow$ *res* : *bool*
- *piso* $\leftarrow$ e.trie ▷ $\mathcal{O}(1)$
- *i* $\leftarrow$ 0 ▷ $\mathcal{O}(1)$
- **while** Esta(c[i],piso) **do** ▷ Se repite a los sumo |c| veces y lo de adentro es $\mathcal{O}(1) \Rightarrow \mathcal{O}(|c|)$
  - *piso* $\leftarrow$ Buscar(c[i],piso).prox ▷ $\mathcal{O}(1)$
  - *i* $\leftarrow$ i++ ▷ $\mathcal{O}(1)$
- *res* $\leftarrow$ (i = |c|)

Complejidad: $\mathcal{O}(|c|)$

Algorithm 55 iEsta

ESTA(**in** *c*: **char**, **in/out** *t*: **diccT(string,**σ**)**) $\rightarrow$ *res* : **bool**
Pre $\equiv$ {esta(c,t)}
Post $\equiv$ {res $=_{obs}$ estaLetra(c,t)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve verdadero si y sólo si el char de entrada esta en alguno de los elementos de la lista.

procedure IESTA(**in** c : **char**, **in** t : **diccT(string,**σ**)**) $\rightarrow$ *res* : *bool*
- *it* $\leftarrow$ crearIr(t) ▷ $\mathcal{O}(1)$
- **while** haySiguiente(it) $\wedge$ Siguiente(it).letra $\neq$ c **do** ▷ Se repite a lo sumo la cantidad de char, que es una constate $\Rightarrow \mathcal{O}(1)$
  - Avanzar(it) ▷ $\mathcal{O}(1)$
- *res* $\leftarrow$ (HaySiguiente(it)) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 56 Buscar

BUSCAR(**in** c: `char`, **in/out** t: `diccT(string,`σ`))` $\rightarrow$ res : `tupla<`*letra* `char,` *significado* `puntero(`σ`),` *prox* `diccT(string,`σ`)>`
Pre $\equiv$ {esta(c,t)}
Post $\equiv$ {alias(res $=_{obs}$ buscar(c,t))}
Complejidad: $\mathcal{O}(1)$
Descripción: Busca el elemento de la lista que este caracterizado con el char de entrada.
Aliasing: El resultado queda ligado con aliasing.

```
procedure BUSCAR(in c : char, in t : diccT(string,σ)) → res : tupla < char, puntero(σ), diccT(string, σ) >
    it ← crearIt(t)
    while Siguiente(it).letra ≠ c do        ▷ Se repite a lo sumo la cantidad de char, que es una constate ⇒ O(1)
        Avanzar(it)                                                                                        ▷ O(1)
    res ← Siguiente(it)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 57 iObtener

```
procedure IOBTENER(in/out e : estr, in c : string) → res : σ
    piso ← e.trie                                                                                          ▷ O(1)
    i ← 0
    while i <|c| - 1 do                ▷ Se repite |c| - 1 veces y lo de adentro es O(1) ⇒ O(|c|)
        piso ← Buscar(c[i],piso).prox                                                                      ▷ O(1)
        i++                                                                                                ▷ O(1)
    res ← *(Buscar(c[i],piso).significado)                                                                 ▷ O(1)
```

Complejidad: $\mathcal{O}(|c|)$

Algorithm 58 iBorrar

```
procedure IBORRAR(in/out e : estr, in c : string)
    Borrar(e.lineal, c)                                                                  ▷ O(#claves(e.lineal))
    piso ← e.trie                                                                                          ▷ O(1)
    i ← 0
    while i <|c|-1 do                  ▷ Se repite |c| - 1 veces y lo de adentro es O(1) ⇒ O(|c|)
        piso ← Buscar(c[i],piso).prox                                                                      ▷ O(1)
        i++                                                                                                ▷ O(1)
    delete(Buscar(s[i],piso).significado)
    Buscar(s[i],piso).significado ← NULL                                                                   ▷ O(1)
```

Comlpejidad: $\mathcal{O}(|c| + \#claves(e.lineal))$

Algorithm 59 iBorrarRapido

```
procedure IBORRARRAPIDO(in/out e : estr, in it : itDiccNom(string,σ))
    EliminarSiguiente(it)                ▷ c es la clave asociada al siguiente del iterador O(|c|)
```

Complejidad: $\mathcal{O}(|clave|)$ donde clave es la clave del siguiente del iterador.

Algorithm 60 i#Claves

```
procedure I#CLAVES(in e : estr)
    res ← #claves(e.lineal)                                                                                ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

2.3.1. Iterador de Diccionario de Nombres

Interfaz

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

se explica con: ITERADOR UNIDIRECCIONAL MODIFICABLE (TUPLA<STRING, σ>)

usa: nat, bool, DiccionarioNombre(string,σ)

géneros: `itDiccNom(string,`σ`)`

Operaciones de DiccionarioNombres(string,σ)

CREARIT(**in** d: `DiccNom(string,`σ`)`) $\rightarrow$ res : `itDiccNom(string,`σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion(secuSuby(res),d)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un iterador al diccionario
Aliasing: El iterador se invalida si se elimina el siguiente sin utilizar la funcion EliminarSiguiente. Los anteriores y siguientes pueden modificarse sin que se invalide el iterador

HAYSIGUIENTE(**in** it: `itDiccNom(string,`σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ hayMas(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve verdadero si y sólo si el iterador tiene un siguiente elemento

SIGUIENTE(**in** it: `itDiccNom(string,`σ`)`) $\rightarrow$ res : `tupla<string,`σ`>`
Pre $\equiv$ {hayMas(it)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ prim(Siguiente(it)))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el siguiente del iterador
Aliasing: res.siguiente es modificable si y solo si el iterador es modificable. res.clave no es modificable

SIGUIENTECLAVE(**in** it: `itDiccNom(string,`σ`)`) $\rightarrow$ res : `string`
Pre $\equiv$ {hayMas(it)}
Post $\equiv$ {res $=_{\text{obs}}$ Π_1(prim(Siguiente(it)))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la siguiente clave
Aliasing: res no es modificable

SIGUIENTESIGNIFICADO(**in** it: `itDiccNom(string,`σ`)`) $\rightarrow$ res : σ
Pre $\equiv$ {hayMas(it)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ Π_2(prim(Siguiente(it))))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el siguiente significado del iterador
Aliasing: res es modificable si y solo si el iterador es modificable

AVANZAR(**in/out** it: `itDiccNom(string,`σ`)`)
Pre $\equiv$ {hayMas(it) $\wedge$ it=it_0}
Post $\equiv$ {it = avanzar(it_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Avanza una posicion el iterador

EliminarSiguiente(**in/out** *it*: `itDiccNom(string,`σ`)`)
Pre $\equiv$ {hayMas(it) $\wedge$ it=it_0}
Post $\equiv$ {it $=_{obs}$ Eliminar(it_0)}
Complejidad: $\mathcal{O}(|c|)$ donde c es la clave asociada al siguiente del iterador
Descripción: Elimina el siguiente del iterador

Representación

Representación del iterador

`itDiccNom(string,`σ`)` **se representa con** `ite`

Donde *ite* es *it* : itDicc(string,σ)

Rep : ite $\longrightarrow$ bool
Rep(i) $\equiv$ true $\iff$ Rep(ite.it)

Abs : ite i $\longrightarrow$ itDiccNom(string,σ) {Rep(i)}
Abs(i) $\equiv$ Abs(i.it)

Algorithm 61 iCrearIt

procedure iCrearIt(`in` e : `estr`) $\rightarrow$ *res* : *ite*
res $\leftarrow$ crearIt(e.lineal) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 62 iHaySiguiente

procedure iHaySiguiente(`in` it : `ite`) $\rightarrow$ *res* : *bool*
res $\leftarrow$ haySiguiente(ite) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 63 iSiguiente

procedure iSiguiente(`in` it : `ite`) $\rightarrow$ *res* : *tupla* $< string, \sigma >$
res $\leftarrow$ Siguiente(ite) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 64 iSiguienteClave

procedure iSiguienteClave(`in` it : `ite`) $\rightarrow$ *res* : *string*
res $\leftarrow \Pi_1$(Siguiente(ite)) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 65 iSiguienteSignificado

procedure iSiguienteSignificado(`in` it : `ite`) $\rightarrow$ *res* : σ
res $\leftarrow \Pi_2$(Siguiente(ite)) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 66 iAvanzar

procedure iAvanzar(`in/out` it : `ite`)
Avanzar(ite) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 67 iEliminarSiguiente

```
procedure iEliminarSiguiente(in it : ite) → res : tupla < string, σ >
    c ← SiguienteClave(ite)                                              ▷ O(1)
    EliminarSiguiente(ite)                                               ▷ O(1)
    piso ← e.trie                                                        ▷ O(1)
    i ← 0                                                                ▷ O(1)
    while i <|c|-1 do          ▷ Se repite |c| - 1 veces y lo de adentro es O(1) ⇒ O(|c|)
        piso ← Buscar(c[i],piso).prox                                    ▷ O(1)
        i++                                                              ▷ O(1)
    delete(Buscar(s[i],piso).significado)
    Buscar(s[i],piso).significado ← NULL                                 ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

2.3.2. Iterador de Claves de Diccionario de Nombres

Interfaz

se explica con: Iterador Unidireccional (string)

usa: nat, bool, DiccionarioNombre(string,σ), IteradorDiccionarioNombres(tupla<string,σ>)

géneros: `itClavesDiccN(string)`

Operaciones de Iterador de Claves de DiccionarioNombres(string)

CrearIt(**in/out** d: `diccNom(string,`σ`)`) $\rightarrow$ res : `itClavesDiccN(string)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ CrearItUni(Secuenciar(claves(d)))}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un iterador no modificable de las claves del diccionario de entrada

Secuenciar : conj(string) $\longrightarrow$ secu(string)

Secuenciar(cs) $\equiv$ **if** $\emptyset$?(cs) **then** <> **else** dameUno(cs) $\bullet$ Secuenciar(sinUno(cs)) **fi**

HaySiguiente(**in** it: `itClavesDiccN(tupla<string,`σ`>)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ HayMas?(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve verdadero si y sólo si el iterador tiene un siguiente elemento

SiguienteClave(**in** it: `itClavesDiccN(tupla<string,`σ`>)`) $\rightarrow$ res : `string`
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ {res $=_{obs}$ Π_1(Actual(it))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la siguiente clave del diccionario
Aliasing: res no es modificable

Avanzar(**in/out** it: `itClavesDiccN(tupla<string,`σ`>)`)
Pre $\equiv$ {HayMas?(it) $\wedge$ it=it_0 }
Post $\equiv$ {it $=_{obs}$ Avanzar(it_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Avanza el iterador

Representación

`itClavesDiccN(string)` **se representa con** `ite`

Donde ite es it : itDiccNom(tupla<string,σ>)

Rep : ite $\longrightarrow$ bool
Rep(i) $\equiv$ true $\Longleftrightarrow$ Rep(ite.it)

Abs : ite i $\longrightarrow$ itClavesDiccN(string) {Rep(i)}

Abs(i) $\equiv$ Abs(i.it)

Algoritmos

Algorithm 68 iCrearIt

procedure ICREARIT(in d : diccNom(string,σ)) $\rightarrow$ *res* : *ite*
res $\leftarrow$ CrearIt(d) // crearIt del iterador de Diccionario Nombre ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 69 iHaySiguiente

procedure IHAYSIGUIENTE(in it : ite) $\rightarrow$ *res* : *bool*
res $\leftarrow$ HaySiguiente(it) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 70 iSiguienteClave

procedure ISIGUIENTECLAVE(in it : ite) $\rightarrow$ *res* : *string*
res $\leftarrow$ SiguienteClave(ite) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 71 iAvanzar

procedure IAVANZAR(in/out it : ite)
Avanzar(ite) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

2.4. Diccionario Logaritmico

Interfaz

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

se explica con: DICCIONARIOACOTADO(NAT,σ)

usa: nat, bool

géneros: `diccLog(nat, σ)`

Operaciones de DiccionarioLog(nat,σ)

VACIO(**in** tam: `nat`) $\rightarrow$ res : `diccLog(nat, σ)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ vacio(tam)}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un diccionario vacío con tam como límite de claves.

DEFINIR(**in/out** d: `diccLog(nat, σ)`, **in** c: `nat`, **in** s: σ) $\rightarrow$ res : `itDiccLog(tupla<nat,σ>)`
Pre $\equiv$ {($\forall$ c' : nat)(c' $\in$ claves(d) $\Rightarrow$ c $>$ c') $\wedge$ d=d_0 $\wedge$ #claves(d) $<$ tamaño(d)}
Post $\equiv$ {d $=_{obs}$ Definir(c,s,d_0) $\wedge$ alias(Siguiente(res) $=_{obs}$ tupla<c,s>)}
Complejidad: $\mathcal{O}(1)$
Descripción: Agrega la clave c con significado s, y devuelve un iterador apuntando a esta nueva difinición.
Aliasing: Del siguiente de res sólo puedo modificarse el significado (si σ es modificable).

DEFINIRLENTO(**in/out** d: `diccLog(nat, σ), in c: nat, in s: σ`) $\rightarrow$ res : `itDiccLog(tupla<nat,σ>)`
Pre $\equiv$ {$\neg$ def(c,d) $\wedge$ d=d_0 $\wedge$ #claves(d) $\leq$ tamaño(d)}
Post $\equiv$ {d $=_{obs}$ definir(c,s,d_0) $\wedge$ alias(Siguiente(res) $=_{obs}$ tupla<c,s>)}
Complejidad: $\mathcal{O}$(#claves(d))
Descripción: Agrega la clave c con significado s, y devuelve un iterador apuntando a esta nueva difinición.
Aliasing: Del siguiente de res sólo puedo modificarse el significado (si σ es modificable).

DEFINIDO(**in/out** d: `diccLog(nat,σ)`, **in** c: `nat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ def?(c,d)}
Complejidad: $\mathcal{O}$(log #claves(d))

OBTENER(**in/out** d: `diccLog(nat,σ)`, **in** c: `nat`) $\rightarrow$ res : σ
Pre $\equiv$ {def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {alias(res $=_{obs}$ obtener(c,d)) $\wedge$ ($\forall$ c' : nat)((c' $\in$ claves(d) $\wedge$ c $\neq$ c') $\Rightarrow_{L}$ obtener(c',d) $=_{obs}$ obtener(c,d_0))}
Complejidad: $\mathcal{O}$(log #claves(d))
Descripción: Devuelve el significado de la clave c.

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

Representación

Representación DiccionarioLog(nat, σ)

`diccLog(nat, σ)` **se representa con** `estr`

donde `estr` es `tupla`(*tamaño*: `nat`
, *valores*: `arreglo dimensionable de tupla <`*clave* `: nat,` *significado* `: σ>` , *proxADefinir*: `nat`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ SinRepetidos(Claves(e.valores)) $\wedge$ Ordenada(Claves(e.valores))
$\wedge$ e.tamaño $=_{obs}$ tamaño(e.valores) $\wedge$ e.proxADefinir $\leq$ e.tamaño
$\wedge$ ($\forall$ i : nat)((i $<$ e.tamaño $\wedge$ $\neg$definido(e.valores, i)) $\Rightarrow_L$
(($\forall$ j : nat)((i $<$ j $\wedge$ j $<$ e.tamaño) $\Rightarrow_L$ $\neg$definido(e.valores, j))))
SinRepetidos : secu(nat) $\longrightarrow$ bool
SinRepetidos(s) $\equiv$ **if** vacio?(s) **then** true **else** $\neg$Esta?(prim(s),fin(s)) $\wedge$ sinRepetidos(fin(s)) **fi**

val es arreglo dimensionable de tupla<*clave* : nat, *significado* : σ>
Claves : val $\longrightarrow$ secu(nat)
Claves(v) $\equiv$ **if** tamaño(v) = 0 **then** <> **else** Primero(v).clave $\bullet$ Claves(fin(v)) **fi**

Ordenada : secu(nat) $\longrightarrow$ bool
Ordenada(s) $\equiv$ **if** vacia?(s) **then** true **else** ($\forall$ n : nat) (n $\in$ fin(s) $\Rightarrow$ n $>$ Primero(s)) $\wedge$ Ordenada(fin(s)) **fi**

Abs : estr e $\longrightarrow$ diccLog(nat,σ) {Rep(e)}
Abs(e) $\equiv$ d : diccLog(nat,σ)/
e.tamaño $=_{obs}$ tamaño(d)
($\forall$ c : nat)(def?(c,d) $\Leftrightarrow$ ($\exists$ i : nat)(0 $\leq$ i $<$ e.tamaño) $\wedge$ definido(e.valores,i) $\wedge_L$ e.valores[i].clave $=_{obs}$ c)
$\wedge_L$ (def(c,d) $\Rightarrow_L$ Obtener(c,d) $=_{obs}$ Significado(c,e))
Significado : nat $\times$ estr $\longrightarrow$ σ {def?(c,d)}
Significado(c,e) $\equiv$ e.valores[Posicion(c,e.valores)].significado

Posicion : nat $\times$ val $\longrightarrow$ nat
Posicion(c,v) $\equiv$ **if** Primero(v).clave = c **then** 0 **else** 1 + Posicion(c,fin(v)) **fi**

Algorithm 72 iVacio

procedure IVACIO(`in` tam : `nat`) $\rightarrow$ *res* : *estr*
e.tamao $\leftarrow$ tam ▷ $\mathcal{O}(1)$
e.valores $\leftarrow$ CrearArreglo(tam) ▷ $\mathcal{O}$(tam)
e.proxADefinir $\leftarrow$ 0 ▷ $\mathcal{O}$(tam)

Complejidad: $\mathcal{O}(1)$

Algorithm 73 iDefinir

procedure IDEFINIR(`in/out` e : `estr`, `in` c : `nat`, `in` s : σ) $\rightarrow$ *res* : *ite*
e.valores[*e.proxADef*].*clave* $\leftarrow$ c ▷ $\mathcal{O}(1)$
e.valores[*e.proxADef*].*signficado* $\leftarrow$ Copiar(s) ▷ $\mathcal{O}$(copy(s))
res.pos $\leftarrow$ e.proxADefinir ▷ $\mathcal{O}(1)$
res.dic $\leftarrow$ puntero(e) ▷ $\mathcal{O}(1)$
e.proxADefinir $\leftarrow$ e.proxADefinir + 1 ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}$(copy(s))

Algorithm 74 iDefinido

procedure iDefinido(in e : estr, in c : nat) → *res* : *bool*
- *arriba* ← e.proxADefinir - 1 ▷ $\mathcal{O}(1)$
- *abajo* ← 0 ▷ $\mathcal{O}(1)$
- **while** arriba > abajo **do** ▷ Se repite log e.proxADefinir veces y lo de adentro es $\mathcal{O}(1)$ ⇒ $\mathcal{O}$(log e.proxADefinir)
  - *medio* ← (arriba + abajo) / 2 ▷ $\mathcal{O}(1)$
  - **if** e.valores[medio].clave < c **then** ▷ $\mathcal{O}(1)$
    - *abajo* ← medio ▷ $\mathcal{O}(1)$
  - **else**
    - *arriba* ← medio ▷ $\mathcal{O}(1)$
- *res* ← (e.valores[medio].clave = c) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}$(log #claves)
Justificación: e.proxADefinir − 1 coincide con la cantidad de claves definidas en el diccionario, pues se van agregando adelante. $\mathcal{O}$(e.proxADefinir) = $\mathcal{O}$(#claves)

Algorithm 75 iDefinirLento

procedure iDefinirLento(in/out e : estr, in c : nat, in s : σ) → *res* : *ite*
- *i* ← e.proxADefinir ▷ $\mathcal{O}(1)$
- **while** i > 0 ∧ e.valores[i-1].clave > c **do** ▷ En el peor caso se repite e.proxADefinir veces, y lo de adentro es $\mathcal{O}$(copy(s)) ⇒ $\mathcal{O}$(copy(s) * e.proxADefinir)
  - *e.valores*[*i*].*clave* ← e.valores[i-1].clave ▷ $\mathcal{O}(1)$
  - *e.valores*[*i*].*significado* ← e.valores[i-1].significado ▷ $\mathcal{O}$(copy(s))
  - *i* ← i − 1 ▷ $\mathcal{O}(1)$
- *e.valores*[*i*].*clave* ← c ▷ $\mathcal{O}(1)$
- *e.valores*[*i*].*significado* ← Copiar(s) ▷ $\mathcal{O}$(copy(s))
- *res.pos* ← e.proxADefinir ▷ $\mathcal{O}(1)$
- *res.dic* ← puntero(e) ▷ $\mathcal{O}(1)$
- *e.proxADefinir* ← e.proxADefinir + 1 ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}$(#claves * copy(s))
Justificación: e.proxADefinir − 1 coincide con la cantidad de claves definidas en el diccionario, pues se van agregando adelante. $\mathcal{O}$(e.proxADefinir) = $\mathcal{O}$(#claves). $\mathcal{O}(1)$ + $\mathcal{O}$(copy(s)) + $\mathcal{O}$(copy(s) * e.proxADefinir) = $\mathcal{O}$(copy(s) * e.proxADefinir) = $\mathcal{O}$(#claves * copy(s))

Algorithm 76 iObtener

procedure iObtener(in e : estr, in c : nat) → *res* : σ
- *arriba* ← e.proxADefinir - 1 ▷ $\mathcal{O}(1)$
- *abajo* ← 0 ▷ $\mathcal{O}(1)$
- **while** arriba > abajo **do** ▷ Se repite log e.proxADefinir veces y lo de adentro es $\mathcal{O}(1)$ ⇒ $\mathcal{O}$(log e.proxADefinir)
  - *medio* ← (arriba + abajo) / 2 ▷ $\mathcal{O}(1)$
  - **if** e.valores[medio].clave < c **then** ▷ $\mathcal{O}(1)$
    - *abajo* ← medio ▷ $\mathcal{O}(1)$
  - **else**
    - *arriba* ← medio ▷ $\mathcal{O}(1)$
- *res* ← e.valores[medio].significado ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}$(log #claves)
Justificación: e.proxADefinir − 1 coincide con la cantidad de claves definidas en el diccionario, pues se van agregando adelante. $\mathcal{O}$(e.proxADefinir) = $\mathcal{O}$(#claves)

2.4.1. Iterador de Diccionario Logaritmico

Interfaz

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

se explica con: ITERADOR UNIDIRECCIONAL MODIFICABLE(TUPLA<NAT,σ>)

usa: nat, bool, diccionarioLog(nat,σ)

géneros: `itDiccLog(tupla<nat, σ>)`

Operaciones de Iterador DiccionarioLog(nat, σ)

HAYSIGUIENTE(**in** it: `itDiccLog(tupla<nat,σ>)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ HayMas?(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve verdadero si y sólo si quedan elementos que recorrer en el iterador.

SIGUIENTECLAVE(**in** it: `itDiccLog(tupla<nat, σ>)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ {res $=_{\text{obs}}$ Π_1(Siguiente(it))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la clave del siguiente elemento del iterador.
Aliasing: res no es modificable.

SIGUIENTESIGNIFICADO(**in/out** it: `itDiccLog(tupla<nat, σ>)`) $\rightarrow$ res : σ
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ Π_2(Siguiente(it)))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el siguiente del siguiente elemento del iterador.
Aliasing: res es modificable si y sólo si σ es modificable.

Representación

`itDiccLog(tupla <nat, σ>)` **se representa con** `ite`

donde `ite` es `tupla(`*pos*: `nat` , *dic*: `puntero(estr)`)
donde *estr* es la representación del diccionarioLog(nat,σ)

Rep : ite $\longrightarrow$ bool
Rep(i) $\equiv$ true $\iff$ i.pos $\leq$ *(i.dic).proxADefinir

Abs : ite i $\longrightarrow$ itDiccLog(tupla<nat,σ>) {Rep(i)}
Abs(i) $=_{\text{obs}}$ itL: itDiccLog(tupla<nat,σ>) | Siguientes(itL) $=_{\text{obs}}$
SecuenciarClavesArreglo(*(i.dic).valores, *(i.dic).tamaño, i.pos)

SecuenciarClavesArreglo : arreglo(tupla<*clave*:nat $\times$ *significado*:σ>) $\times$ nat $\longrightarrow$ secu(nat)

SecuenciaClavesArreglo(var, tam, pos) $\equiv$ **if** pos $=_{obs}$ tam **then**
<>
else
var[pos].clave • SecuenciarArreglo(var, tam, pos + 1)
fi

Algoritmos

Algorithm 77 iHaySiguiente

procedure iHaySiguiente(in i : ite) $\rightarrow$ *res* : *bool*
res $\leftarrow$ (i.pos < *(i.dic).tamaño) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 78 iSiguienteClave

procedure iSiguienteClave(in i : ite) $\rightarrow$ *res* : *nat*
res $\leftarrow$ *(i.dic).valores[i.pos].clave ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 79 iSiguienteSignificado

procedure iSiguienteSignificado(in i : ite) $\rightarrow$ *res* : *nat*
res $\leftarrow$ *(i.dic).valores[i.pos].significado ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

2.4.2. Iterador de Claves de Diccionario Logaritmico

Interfaz

se explica con: Iterador Unidireccional (nat)

usa: nat, bool, DiccionarioLog(nat,σ)

géneros: `itClavesDiccLog(nat)`

Operaciones de Iterador de Claves de DiccionarioPlacas(nat)

CrearIt(**in** d: `diccLog(nat,`σ`)`) $\rightarrow$ *res* : `itClavesDiccLog(nat)`
Pre $\equiv$ {true}
Post $\equiv$ {Permutacion(Siguientes(res), Secuenciar(claves(d)))}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un iterador no modificable de las claves del diccionario de entrada. El iterador se invalida si se modifica el diccionario.

Secuenciar : conj(string) $\longrightarrow$ secu(string)

Secuenciar(cs) $\equiv$ **if** $\emptyset$?(cs) **then** <> **else** dameUno(cs) • Secuenciar(sinUno(cs)) **fi**

Permutacion : secu(nat) $\times$ secu(nat) $\longrightarrow$ bool

Permutacion(s,t) $\equiv$ Iguales(MultiConjuntizar(s), MultiConjuntizar(t))

MultiConjuntizar : secu(nat) $\longrightarrow$ multiconj(nat)

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

Iguales : multiconj(nat) × multiconj(nat) $\longrightarrow$ bool

Iguales(m_1, m_2) $\equiv$ **if** $\#m_1 = 0$ **then**
$(\#m_2 = 0)$
else
$\#$(dameUno(m_1),m_1) = $\#$(dameUno(m_1),m_2)
$\wedge$ Iguales(sinUno(m_1), m_2 −{dameUno(m_1)})
fi

HaySiguiente(**in** *it*: `itClavesDiccLog(nat)`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ HayMas?(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve verdadero si y sólo si el iterador tiene un siguiente elemento

SiguienteClave(**in** *it*: `itClavesDiccLog(nat)`) → *res* : `string`
Pre ≡ {HayMas?(it)}
Post ≡ {res =$_{obs}$ Π_1(Actual(it))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la siguiente clave del diccionario.
Aliasing: res no es modificable.

Avanzar(**in/out** *it*: `itClavesDiccLog(nat)`)
Pre ≡ {HayMas?(it) $\wedge$ it=it_0 }
Post ≡ {it =$_{obs}$ Avanzar(it_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Avanza el iterador.

Representación

`itClavesDiccLog(nat)` **se representa con** `ite`

donde `ite` es `tupla`(*pos*: `nat` , *dic*: `puntero(estr)`)
donde *estr* es la representación del diccionarioLog

Rep : ite $\longrightarrow$ bool
Rep(i) $\equiv$ true $\iff$ i.pos $\leq$ *(i.dic).proxADefinir $\wedge$ i.dic $\neq$ NULL

Abs : ite i $\longrightarrow$ itClavesDiccLog(nat) {Rep(i)}
Abs(i) =$_{obs}$ itC: itClavesDiccLog(nat) |
Siguientes(itC) =$_{obs}$ SecuenciarClavesArreglo(*(i.dic).valores,
*(i.dic).tamaño, i.pos)

SecuenciarClavesArreglo : arreglo(tupla<*clave*:nat × *significado*:σ>) × nat $\longrightarrow$ secu(nat)
SecuenciaClavesArreglo(var, tam, pos) $\equiv$ **if** pos =$_{obs}$ tam **then**
<>
else
var[pos].clave • SecuenciarArreglo(var, tam, pos + 1)
fi

Algoritmos

Algorithm 80 iCrearIt

procedure iCrearIt(`in` e : `estr`) → *res* : *ite*
res.pos ← 0 ▷ $\mathcal{O}(1)$
res.dic ← puntero(e) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 81 iHaySiguiente

procedure IHAYSIGUIENTE(in i : ite) → *res* : *bool*
res ← (i.pos < *(i.dic).tamaño) ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 82 iSiguienteClave

procedure ISIGUIENTECLAVE(in i : ite) → *res* : *nat*
res ← *(i.dic).valores[i.pos] ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Algorithm 83 iAvanzar

procedure IAVANZAR(in/out i : ite)
i.pos ← i.pos + 1 ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

2.5. Diccionario$_H$

Interfaz

parámetros formales
géneros σ

usa: nat, bool, arreglo Dimensionable

géneros: `dicch(nat,`σ`)`

se explica con: DICCIONARIOACOTADO(NAT,σ)

Operaciones de Dicch(nat,σ)

VACIO(**in** *cant*: `nat`) $\rightarrow$ *res* : `dicch(nat, `σ`)`
Pre $\equiv$ {cant $>$ 0}
Post $\equiv$ {res $=_{obs}$ vacio(cant)}
Complejidad: $\mathcal{O}$(cant)
Descripción: Crea un diccionario vacio de cant posiciones.

DEFINIR(**in** *clave*: `nat`, **in** *significado*: σ, **in/out** *dicc*: `dicch(nat,`σ`)`)
Pre $\equiv$ {dicc=$dicc_0$ $\wedge$ $\neg$def?(clave,$dicc_0$) $\wedge$ claves($dicc_0$) $<$tamaño($dicc_0$)}
Post $\equiv$ {dicc $=_{obs}$ Definir(clave,significado,$dicc_0$)}
Complejidad: $\mathcal{O}$(1) en caso promedio
Descripción: Agrega un elemento al diccionario.

OBTENER(**in** *clave*: `nat`, **in/out** *dicc*: `dicch(nat,`σ`)`) $\rightarrow$ *res* : σ
Pre $\equiv$ {def?(clave,$dicc_0$) $\wedge$ dicc=$dicc_0$}
Post $\equiv$ {alias(res $=_{obs}$ obtener(clave,dicc)) $\wedge$
($\forall$ clave' : nat)((clave' $\in$ claves(dicc) $\wedge$ clave $\neq$ clave')$\Rightarrow_L$ obtener(clave',dicc) $=_{obs}$ obtener(clave',$dicc_0$))}
Complejidad: $\mathcal{O}$(1) en caso promedio

Representación

Representacion del DiccionaroHash(nat, σ)

`dicch(nat,`σ`)` **se representa con** `estr`

donde `estr` es `tupla(`*cant*: `nat` , *valores*: `Adt`)
Donde Adt es arreglo Dimensionable de tupla<*clave* : nat, *significado* : σ>)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ e.cant $=_{obs}$ Tamaño(e.valores) $\wedge_L$
($\forall$ i, j : nat) (i $<$ e.cant $\wedge$ j $<$ e.cant $\wedge$ i $\neq$ j $\wedge$ definido(e.valores,i) $\wedge$ definido(e.valores,j))
$\Rightarrow_L$ e.valores[i].clave $\neq$ e.valores[j].clave

Abs : estr e $\longrightarrow$ dicch(nat, σ) {Rep(e)}
Abs(e) $\equiv$ dic : dicch(nat, σ) /
($\forall$ clave : nat)
(Def?(clave,dic) $\Leftrightarrow$ ($\exists$ i : nat) (0 $\leq$ i $<$ e.cant $\wedge$ Definido(e.valores, i) $\wedge_L$ e.valores[i].clave = clave))
$\wedge$ (Def?(clave,dic)
$\Rightarrow_L$ Obtener(clave,dic) $=_{obs}$ e.valores[mod(hash(clave, e) + corrimiento(hash(clave, e), e) , cant)].significado

hash : nat $\times$ estr $\longrightarrow$ nat
hash(clave, e) $\equiv$ mod(clave, e.cant)

corrimiento : nat $\times$ estr $\longrightarrow$ nat

corrimiento(clave, e) $\equiv$ **if** e.valores[mod(clave, e.cant)].clave = clave **then**
0
else
1 + corrimiento(clave + 1, e)
fi

Algoritmos

Algorithm 84 iVacio

procedure IVACIO(in n : nat) $\rightarrow$ *res* : estr
var e : estr
$e.cant \leftarrow$ n ▷ $\Theta(1)$
$e.valores \leftarrow$ CrearArreglo(n) ▷ $\mathcal{O}$(n)
$res \leftarrow$ e ▷ $\Theta(1)$

Complejidad: $\mathcal{O}$(n)

Algorithm 85 iDefinir

procedure IDEFINIR(in clave : nat, in significado : σ, in/out e : estr)
var h : nat
$h \leftarrow$ mod(clave, e.cant) ▷ $\mathcal{O}(1)$
while Definido(e.valores, h) **do** ▷ En promedio no se entra al ciclo
$h \leftarrow$ mod(h+1, e.cant) ▷ $\mathcal{O}(1)$
$e.valores[h] \leftarrow$ tupla<clave, significado> ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$ promedio
Justificación: sabiendo que hay una distribución uniforme de 'claves', podemos asumir que en promedio no se entraría al ciclo, cerrando en promedio una suma de operaciones de complejidad $\mathcal{O}(1)$.

Algorithm 86 iObtener

procedure IOBTENER(in clave : nat,in/out e : estr) $\rightarrow$ *res* : σ
var h : nat
$h \leftarrow$ mod(clave, e.cant) ▷ $\mathcal{O}(1)$
while e.valores[h].clave $\neq$ clave **do** ▷ En promedio no se entra al ciclo
$h \leftarrow$ mod(h+1, e.cant) ▷ $\mathcal{O}(1)$
$res \leftarrow$ e.valores[h].significado ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$ promedio
Justificación: Misma justificación que el algoritmo anterior

2.6. Posicion

Interfaz

se explica con: TAD Posicion.

géneros: `posicion`.

Operaciones básicas de posicion

CrearPosicion(**in** x: `nat`,**in** y: `nat`) $\rightarrow$ res : `posicion`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ <x,y>}
Complejidad: $\Theta(1)$
Descripción: Crea una nueva instancia de posición haciendo copia por valor de los parámetros de entrada.

X(**in** p: `posicion`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {alias($res =_{obs}$ X(p))}
Complejidad: $\Theta(1)$
Descripción: Devuelve el valor de la variable "X"de la posición
Aliasing: Hay aliasing entre el resultado de la función y el valor "X"de la instancia posicion.

Y(**in** p: `posicion`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {alias($res =_{obs}$ Y(p))}
Complejidad: $\Theta(1)$
Descripción: Devuelve el valor de la variable "Y"de la posicion
Aliasing: Hay aliasing entre el resultado de la función y el valor "Y"de la instancia posicion.

SumarPosiciones(**in** p_1 : `posicion`,**in** p_2 : `posicion`) $\rightarrow$ res : `posicion`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ <X(p_1) + X(p_2), Y(p_1) + Y(p_2)>}
Complejidad: $\Theta(1)$
Descripción: Devuelve una nueva instancia del tipo posicion, resultado de sumar ambas X's e Y's.

RestarPosiciones(**in** p_1 : `posicion`,**in** p_2 : `posicion`) $\rightarrow$ res : `posicion`
Pre $\equiv$ {Y(p_2) < Y(p_1) $\wedge$ X(p_2) < X(p_1)}
Post $\equiv$ {$res =_{obs}$ <X(p_1) − X(p_2), Y(p_1) − Y(p_2)>}
Complejidad: $\Theta(1)$
Descripción: Devuelve una nueva instancia del tipo posicion, resultado de restar a la primer posicion, la segunda.

DistanciaPosiciones(**in** p_1 : `posicion`,**in** p_2 : `posicion`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ Max(X(p_1), X(p_2)) - Min(X(p_1),X(p_2)) + Max(Y(p_1),Y(p_2)) - Max(Y(p_1),Y(p_2)) }
Complejidad: $\Theta(1)$
Descripción: Devuelve el resultado de calcular la distancia entre dos instancias de posicion, en base a sus observadores X e Y.

Representación

Representación de posicion

`posicion` **se representa con** `estr`

donde `estr` es `tupla`(x: `nat`, y: `nat`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$ true

Abs : estr e $\longrightarrow$ posicion {Rep(e)}

Abs(e) $=_{\text{obs}}$ p: posicion | X(p) $= e$.x $\wedge$ Y(p) $= e$.y

Algoritmos

Algorithm 87 iCrearPosicion

procedure iCrearPosicion(in x : nat, in y : nat) $\rightarrow$ *res* : estr
var pos : estr
pos.x $\leftarrow$ x ▷ $\Theta(1)$
pos.y $\leftarrow$ y ▷ $\Theta(1)$
res $\leftarrow$ pos ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Algorithm 88 iX

procedure iX(in pos : estr) $\rightarrow$ *res* : *nat*
res $\leftarrow$ pos.x ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Algorithm 89 iY

procedure iY(in pos : estr) $\rightarrow$ *res* : *nat*
res $\leftarrow$ pos.y ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Algorithm 90 iSumarPosiciones

procedure iSumarPosicion(in pos_1 : estr, in pos_2 : estr) $\rightarrow$ *res* : estr
var pos : estr
pos $\leftarrow$ iCrearPosicion(pos_1.x + pos_2.x, pos_1.y + pos_2.y) ▷ $\Theta(1)$
res $\leftarrow$ pos ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Algorithm 91 iRestarPosiciones

procedure iRestarPosiciones(in pos_1 : estr, in pos_2 : estr) $\rightarrow$ *res* : *estr*
var pos : estr
pos $\leftarrow$ iCrearPosicion(pos_1.x − pos_2.x, pos_1.y − pos_2.y) ▷ $\Theta(1)$
res $\leftarrow$ pos ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Algorithm 92 iDistanciaPosiciones

procedure iDistanciaPosiciones(in pos_1 : estr, in pos_2 : estr) $\rightarrow$ *res* : *nat*
var pos : nat
pos $\leftarrow$ Max(pos_1.x, pos_2.x) − Min(pos_1.x, pos_2.x) + Max(pos_1.y, pos_2.y) − Min(pos_1.y,pos_2.y) ▷ $\Theta(1)$
res $\leftarrow$ pos ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

2.7. Matriz(α)

Interfaz

parámetros formales
géneros α
función COPIAR(**in** a: α) $\rightarrow$ 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: MATRIZ(α).

géneros: `matriz`(α).

Operaciones básicas de matriz

CREARMATRIZ(**in** *alto*: `nat`, **in** *ancho*: `nat`, **in** a: α) $\rightarrow$ res : `matriz`(α)
Pre $\equiv$ {ancho $>$ 0 $\wedge$ alto $>$ 0}
Post $\equiv$ {res $=_{\text{obs}}$ Crear(ancho, alto, a)}
Complejidad: $\mathcal{O}(ancho$ x $alto$ x copy(σ))
Descripción: Genera una matriz de *ancho***alto* que contenga todos a.

DEFINIR(**in/out** m: `matriz`(α), `in` *ancho*: `nat`, `in` *alto*: `nat`, `in` a: α)
Pre $\equiv$ {posValida(m, ancho, alto) $\wedge$ m_0=m}
Post $\equiv$ {m $=_{\text{obs}}$ definir(alto, ancho, a, m_0)}
Complejidad: $\mathcal{O}$(copy(α))
Descripción: Defino el elemento a en la matriz.

•[•][•](**in/out** m: `matriz`(α), **in** *ancho*: `nat`, **in** *alto*: `nat`) $\rightarrow$ res : α
Pre $\equiv$ {posValida(m, ancho, alto)}
Post $\equiv$ {alias($res =_{\text{obs}}$ valor(m, ancho, alto))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el elemento en la posicion *ancho***alto* de la matriz m con alias.
Aliasing: res es modificable si y sólo si α es modificable.

COPIAR(**in** m: `matriz`(α)) $\rightarrow$ res : `matriz`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ m}
Complejidad: $\mathcal{O}(ancho$ x $alto$ x copy(α))
Descripción: Crea una copia de la matriz m.

ALTO(**in** m: `matriz`(α)) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ alto(m)}
Complejidad: $\mathcal{O}(1)$
Descripción: Retorna la altura de la matriz m.

ANCHO(**in** m: `matriz`(α)) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ ancho(m)}
Complejidad: $\mathcal{O}(1)$
Descripción: Retorna el ancho de la matriz m.

POSVALIDA(**in** m: `matriz`(α), **in** *ancho*: `nat`, **in** *alto*: `nat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ posValida(m, ancho, alto)}
Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve si es una posición válida de mi matriz o no.

Representación

Representación de la matriz

`matriz(`α`)` se representa con `estr`

donde `estr` es `tupla(`*matriz*: `arreglo_dimensionable de arreglo_dimensionable de` α , *ancho*: `nat` , *alto*: `nat`)

Rep : estr $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ e.ancho $> 0 \wedge$ e.alto $> 0 \wedge$ tamaño(e.matriz) $=$ e.ancho $\wedge$
(∀ i : nat)(i $<$ e.ancho $\Rightarrow_{\text{L}}$ tamaño(e.matriz[i]) $=$ e.alto)

Abs : estr e $\longrightarrow$ matriz(α) {Rep(e)}
Abs(e) $\equiv$ mat: matriz(α) | e.ancho = Ancho(mat) $\wedge$ e.alto = Alto(mat) $\wedge_{\text{L}}$
(∀ i, j: nat)((i $<$ e.ancho $\wedge$ j $<$ e.alto) $\Rightarrow_{\text{L}}$ (e.matriz[i])[j] = Valor?(mat, i, j))

Algoritmos

Algorithm 93 iCrearMatriz

```
1: procedure iCrearMatriz(in ancho : nat, in alto : nat, in a : α) → res : estr
2:     res ← CrearArreglo(ancho) de arreglos_dimensionables de α          ▷ O(ancho)
3:     i ← 0                                                              ▷ O(1)
4:     while i < ancho do          ▷ O(alto x copy(α)) ancho de veces ⇒ O(ancho x alto x copy(α))
5:         j ← 0                                                          ▷ O(1)
6:         res[i] ← CrearArreglo(alto) de α                               ▷ O(alto)
7:         while j < alto do                ▷ O( copy(α) ) alto de veces ⇒ O(alto x copy(α))
8:             res[i][j] ← Copiar(a)                                      ▷ O( copy(α))
9:             j ← j+1                                                    ▷ O(1)
10:        i ← i+1                                                        ▷ O(1)
```

Complejidad: $\mathcal{O}$(ancho x alto x copy(α))
Justificacion: $\mathcal{O}$(ancho)+$\mathcal{O}$(1)+$\mathcal{O}$(ancho x alto x copy(α)) $\Rightarrow$ $\mathcal{O}$(ancho x alto x copy(α))

Algorithm 94 iDefinir

```
1: procedure iDefinir(in/out e : estr, in ancho : nat, in alto : nat, in a : α)
2:     (e.matriz[ancho])[alto] ← Copiar(a)          ▷ O(1) ⇒ O(copy(α))
```

Complejidad: $\mathcal{O}(copy(\alpha))$

Algorithm 95 •[•][•]

```
1: procedure •[•][•](in/out e : estr, in ancho : nat, in alto : nat) → res : α
2:     res ← (e.matriz[ancho])[alto]                          ▷ O(1) ⇒ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 96 iCopiar

```
1: procedure iCopiar(in e : estr) → res : estr
2:     res ← CrearMatriz(e.alto, e.ancho, e[0][0])          ▷ O(e.ancho x e.alto x copy(α))
3:     i ← 0                                                ▷ O(1)
4:     while i < e.ancho do   ▷ O(alto x copy(α)) e.ancho de veces ⇒ O(e.ancho x e.alto x copy(α))
5:         j ← 0                                            ▷ O(1)
6:         while j < e.alto do   ▷ O(copy(α)) e.alto de veces ⇒ O(e.alto x copy(α))
7:             Definir(res, i, j, Copiar(e.matriz[i][j]))   ▷ O(1)+O(copy(α)) ⇒ O(copy(α))
8:             j ← j+1                                      ▷ O(1)
9:         i ← i+1                                          ▷ O(1)
```

Complejidad: $\mathcal{O}$(ancho x alto x $copy(\alpha)$)
Justificacion: Llamo α a el tipo de elemento de e.matriz. $\mathcal{O}$(e.ancho x e.alto x copy(α)) + $\mathcal{O}(1)$ + $\mathcal{O}$(e.ancho x e.alto x $copy(\alpha)$) = $\mathcal{O}$(ancho x alto x $copy(\alpha)$).

Algorithm 97 iAlto

```
1: procedure iAlto(in e : estr) → res : nat
2:     res ← e.alto                                         ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 98 iAncho

```
1: procedure iAncho(in e : estr) → res : nat
2:     res ← e.ancho                                        ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

Algorithm 99 iPosValida

```
1: procedure iPosValida(in e : estr, in ancho : nat, in alto : nat) → res : bool
2:     ancho < e.ancho ∧ alto < e.alto                      ▷ O(1)
```

Complejidad: $\mathcal{O}(1)$

3. Tads

3.1. Posicion

TAD POSICION

géneros posicion

exporta todo

usa Nat

igualdad observacional

$$(\forall p, p' : \text{posicion}) \left(p =_{\text{obs}} p' \iff \begin{pmatrix} \text{X}(p) =_{\text{obs}} \text{X}(p') \\ \wedge\ \text{Y}(p) =_{\text{obs}} \text{Y}(p') \end{pmatrix} \right)$$

observadores básicos

X	: posicion	$\longrightarrow$	Nat
Y	: posicion	$\longrightarrow$	Nat

generadores

<•, •>	: Nat × Nat	$\longrightarrow$	posicion

otras operaciones

• + •	: posicion × posicion	$\longrightarrow$	posicion	
• - •	: posicion p × posicion p'	$\longrightarrow$	posicion	$\{\text{Y}(p') < \text{Y}(p) \wedge \text{X}(p') < \text{X}(p)\}$
Dist	: posicion × posicion	$\longrightarrow$	posicion	

axiomas

X(<x, y>)	$\equiv$	x
Y(<x, y>)	$\equiv$	y
p + p'	$\equiv$	<X(p) + X(p'), Y(p) + Y(p') >
p − p'	$\equiv$	<X(p) − X(p'), Y(p) − Y(p') >
Dist(p,p')	$\equiv$	Max(X(p), X(p')) − Min(X(p), X(p')) + Max(Y(p), Y(p')) − Min(Y(p), Y(p'))

Fin TAD

3.2. Matriz

TAD MATRIZ(α)

igualdad observacional

$$(\forall m, m' : \text{matriz}(\alpha)) \left(m =_{\text{obs}} m' \iff \begin{pmatrix} \text{ancho(m)} =_{\text{obs}} \text{ancho(m')} \wedge \text{alto(m)} =_{\text{obs}} \text{alto(m')} \\ \wedge_{\text{L}} \ (\ (\forall col, fila\text{: nat}\ )\ \text{posValida(m, col, fila}\ ) \Rightarrow_{\text{L}} \\ (\text{valor(m, col, fila)} =_{\text{obs}} \text{valor(m', col, fila}\ )\ )\ ) \end{pmatrix} \right)$$

parámetros formales
géneros α

géneros matriz(α)

exporta matriz(α), generadores, observadores, posValida

usa α, BOOL, NAT

observadores básicos

ancho	: matriz(α)	$\longrightarrow$	nat	
alto	: matriz(α)	$\longrightarrow$	nat	
valor	: matriz(α) m $\times$ columna nat $\times$ fila nat	$\longrightarrow$	α	{posValida(m, columna, fila)}

generadores

crear	: ancho nat $\times$ alto nat $\times$ α	$\longrightarrow$	matriz(α)	{ancho $>$ 0 $\wedge$ alto $>$ 0}
definir	: columna nat $\times$ fila nat $\times$ α $\times$ matriz(α) m	$\longrightarrow$	matriz(α)	{posValida(m, columna, fila)}

otras operaciones

posValida	: matriz(α) $\times$ columna $\times$ fila	$\longrightarrow$	bool

axiomas $\forall m$: matriz(α), $\forall an, al$: nat, $\forall a$: α

ancho(crear(an, al, a))	$\equiv$	an
ancho(definir(c, f, a, m))	$\equiv$	ancho(m)
alto(crear(an, al, a))	$\equiv$	al
alto(definir(c, f, a, m))	$\equiv$	alto(m)
valor(crear(an, al, a), c, f)	$\equiv$	a
valor(definir(c, f, a, m), c', f')	$\equiv$	**if** $f = f' \wedge c = c'$ **then** a **else** valor(m,c',f') **fi**
posValida(m, c, f)	$\equiv$	$f <$ alto(m) $\wedge$ $c <$ ancho(m)

Fin TAD

3.3. Diccionario Acotado

TAD DiccionarioAcotado(clave, significado)

igualdad observacional

$$(\forall d, d' : \text{diccAcot}(\kappa, \sigma)) \left(d =_{\text{obs}} d' \iff \begin{pmatrix} \text{tamaño(d)} =_{\text{obs}} \text{tamaño(d')} \wedge \\ (\forall c : \kappa)(\text{def?}(c, d) =_{\text{obs}} \text{def?}(c, d') \wedge_{\text{L}} \\ (\text{def?}(c, d) \Rightarrow_{\text{L}} \text{obtener}(c, d) =_{\text{obs}} \text{obtener}(c, d')) \end{pmatrix} \right)$$

parámetros formales

géneros clave, significado

géneros diccAcot(clave, significado)

exporta diccAcot(clave, significado), generadores, observadores, borrar, claves

usa Bool, Nat, Conjunto(clave)

observadores básicos

def?	: clave × diccAcot(clave, significado)	$\longrightarrow$	bool	
obtener	: clave c × diccAcot(clave, significado) d	$\longrightarrow$	significado	{def?(c, d)}
tamaño	: diccAcot(clave × significado)	$\longrightarrow$	nat	

generadores

vacío	: nat	$\longrightarrow$	diccAcot(clave, significado)	
definir	: clave × significado × diccAcot(clave, significado) d	$\longrightarrow$	diccAcot(clave, significado)	{#claves(d) < tamaño(d)}

otras operaciones

borrar	: clave c × diccAcot(clave, significado) d	$\longrightarrow$	diccAcot(clave, significado)	{def?(c,d)}
claves	: diccAcot(clave, significado)	$\longrightarrow$	conj(clave)	

axiomas $\forall d$: diccAcot(clave, significado), $\forall c, k$: clave, $\forall s$: significado

def?(c,vacío(n))	$\equiv$	false
def?(c, definir(k, s, d))	$\equiv$	$c = k \vee$ def?(c, d)
obtener(c, definir(k, s, d))	$\equiv$	**if** $c = k$ **then** s **else** obtener(c, d) **fi**
tamaño(vacio(n))	$\equiv$	n
tamaño(definir(k, s, d))	$\equiv$	tamaño(d)

borrar(c, definir(k, s, d))	$\equiv$	**if** $c = k$ **then**
		  if def?(c,d) **then** borrar(c,d) **else** d **fi**
		else
		  definir(k, s, borrar(c, d))
		fi

claves(vacío(n))	$\equiv$	$\emptyset$
claves(definir(c,s,d))	$\equiv$	Ag(c, claves(d))

Fin TAD