Algoritmos y Estructuras de Datos II

Trabajo Práctico 2

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

Primer Cuatrimestre de 2015

Grupo 16

Apellido y Nombre	LU	E-mail
Fernando Frassia	340/13	ferfrassia@gmail.com
Rodrigo Seoane Quilne	910/11	seoane.raq@gmail.com
Sebastian Matias Giambastiani	916/12	sebastian.giambastiani@hotmail.com

Reservado para la cátedra

Instancia	Docente que corrigió	Calificación
Primera Entrega		
Recuperatorio		

Índice

1. Tad Extendidos

1.1. Secu(α)

otras operaciones
elemDeSecu : Secu(α) s × Nat n $\longrightarrow$ RUR $\{n < \text{long}(s)\}$

axiomas
elemDeSecu(s, n) $\equiv$ **if** $n = 0$ **then** prim(s) **else** elemDeSecu(fin(s), n-1) **fi**

1.2. Mapa

observadores básicos
restricciones : Mapa m $\longrightarrow$ secu(restriccion)
nroConexion : estacion e_1 × estacion e_2 × Mapa m $\longrightarrow$ nat$\{e_1, e_2 \subset \text{estaciones(m)} \wedge_{\text{L}} \text{conectadas?}(e_1, e_2, m)\}$

axiomas
restricciones(vacio) $\equiv$ $\langle\ \rangle$
restricciones(agregar(e, m)) $\equiv$ restricciones(m)
restricciones(conectar(e_1, e_2, r, m)) $\equiv$ restricciones(m) $\circ$ r
nroConexion(e_1, e_2,conectar(e_3, e_4, m)) $\equiv$ **if** $((e_1 = e_3 \wedge e_2 = e_4) \vee (e_1 = e_4 \wedge e_2 = e_3))$ **then**
long(restricciones(m)) - 1
else
nroConexion(e_1, e_2, m) - 1
fi
nroConexion(e_1, e_2,agregar(e, m)) $\equiv$ nroConexion(e_1, e_2, m)

2. Red

2.1. Interfaz

Interfaz

se explica con: Red, Iterador Unidireccional(α).

géneros: `red, itConj(Compu)`.

Operaciones básicas de Red

Computadoras(**in** r: `red`) $\rightarrow$ res : `itConj(Compu)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearIt(computadoras(r))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve las computadoras de red.

Conectadas?(**in** r: `red`, **in** c_1: `compu`, **in** c_2: `compu`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$\{c_1, c_2\} \subseteq$ computadoras(r)}
Post $\equiv$ {$res =_{\text{obs}}$ conectadas?(r, c_1, c_2)}
Complejidad: $\mathcal{O}(|c_1| + |c_2|)$
Descripción: Devuelve el valor de verdad indicado por la conexión o desconexión de dos computadoras.

InterfazUsada(**in** r: `red`, **in** c_1: `compu`, **in** c_2: `compu`) $\rightarrow$ res : `interfaz`
Pre $\equiv$ {$\{c_1, c_2\} \subseteq$ computadoras(r) $\wedge_{\text{L}}$ conectadas?(r, c_1, c_2)}
Post $\equiv$ {$res =_{\text{obs}}$ interfazUsada(r, c_1, c_2)}
Complejidad: $\mathcal{O}(|c_1| + |c_2|)$
Descripción: Devuelve la interfaz que c_1 usa para conectarse con c_2

IniciarRed() $\rightarrow$ res : `red`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ iniciarRed()}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea una red sin computadoras.

AgregarComputadora(**in/out** r: `red`, **in** c: `compu`)
Pre $\equiv$ {$r_0 =_{\text{obs}} r \wedge \neg(c \in$computadoras($r$))}
Post $\equiv$ {$r =_{\text{obs}}$ agregarComputadora(r_0, c)}
Complejidad: $\mathcal{O}(|c|)$
Descripción: Agrega una computadora a la red.

Conectar(**in/out** r: `red`, **in** c_1: `compu`, **in** i_1: `interfaz`, **in** c_2: `compu`, **in** i_2: `interfaz`)
Pre $\equiv$ {$r_0 =_{\text{obs}}$ r $\wedge$ $\{c_1, c_2\} \subseteq$ computadoras(r) $\wedge$ ip(c_1) $\neq$ ip(c_2) $\wedge_{\text{L}} \neg$ conectadas?(r, c_1, c_2) $\wedge \neg$ usaInterfaz?(r, c_1, i_1) $\wedge \neg$ usaInterfaz?(r, c_2, i_2)}
Post $\equiv$ {$r =_{\text{obs}}$ conectar(r, c_1, i_1, c_2, i_2)}
Complejidad: $\mathcal{O}(|c_1| + |c_2|)$
Descripción: Conecta dos computadoras y les añade la interfaz correspondiente.

Vecinos(**in** r: `red`, **in** c: `compu`) $\rightarrow$ res : `conj(compu)`
Pre $\equiv$ {$c \in$ computadoras(r)}
Post $\equiv$ {$res =_{\text{obs}}$ vecinos(r, c)}
Descripción: Devuelve todas las computadoras que están conectadas directamente con c

UsaInterfaz?(**in** r: `red`, **in** c: `compu`, **in** i: `interfaz`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$c \in$ computadoras(r)}
Post $\equiv$ {$res =_{\text{obs}}$ usaInterfaz?(r, c, i)}
Descripción: Verifica que una computadora use una interfaz

CaminosMinimos(**in** r: `red`, **in** c_1: `compu`, **in** c_2: `compu`) $\rightarrow$ res : `itConj(`α`)`
Pre $\equiv$ {$\{c_1, c_2\} \subseteq$ computadoras(r)}

Post $\equiv \{res =_{\text{obs}} \text{crearItBi(caminosMinimos(r, } c_1\text{, } c_2\text{))}\}$
Descripción: Devuelve todos los caminos minimos de conexiones entre una computadora y otra

HayCamino?(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `bool`
Pre $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$
Post $\equiv \{res =_{\text{obs}} \text{hayCamino?(r, } c_1\text{, } c_2\text{)}\}$
Descripción: Verifica que haya un camino de conexiones entre una computadora y otra

2.2. Auxiliares

Operaciones auxiliares

CalcularCaminosMinimos(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `conj(lista)`
Pre $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$
Post $\equiv \{res =_{\text{obs}} \text{caminosMinimos(r, } c_1\text{, } c_2\text{)}\}$
Complejidad: $\mathcal{O}(ALGO)$
Descripción: Devuelve los caminos minimos entre c_1 y c_2
CaminosImportantes(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`, **in** *parcial* : `lista`) $\rightarrow$ res : `conj(lista)`
Pre $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$
Post $\equiv \{res =_{\text{obs}} \text{caminosMinimos(r, } c_1\text{, } c_2\text{)}\}$
Complejidad: $\mathcal{O}(ALGO)$
Descripción: Devuelve los caminos suficientes (no todos) para calcular los caminos mínimos entre c_1 y c_2

2.3. Representacion

Representación

`red` **se representa con** `e_red`

donde `e_red` es `tupla`(*vecinosEInterfaces*: `diccString`(*compu*: `string`, tupla(*interfaces*: `diccString`(*compu*: `string`, *interfaz*: `nat`), *compusVecinas*: `conj(compu)`))
, *deOrigenADestino*: `diccString`(*compu*: `string`, `diccString`(*compu*: `string`, *caminosMinimos*: `conj(lista(compu))`))
, *computadoras*: `conj(compu)`)

2.4. InvRep y Abs

1. El conjunto de claves de "uniones" es igual al conjunto de estaciones "estaciones".
2. "#sendas" es igual a la mitad de las horas de "uniones".
3. Todo valor que se obtiene de buscar el significado del significado de cada clave de "uniones", es igual el valor hallado tras buscar en "uniones" con el sinificado de la clave como clave y la clave como significado de esta nueva clave, y no hay otras hojas ademas de estas dos, con el mismo valor.
4. Todas las hojas de "uniones" son mayores o iguales a cero y menores a "#sendas".
5. La longitud de "sendas" es mayor o igual a "#sendas".

Rep : e_mapa $\longrightarrow$ bool
Rep(m) $\equiv$ true $\Longleftrightarrow$

m.estaciones = claves(m.uniones) $\wedge$	1.
m.#sendas = #sendasPorDos(m.estaciones, m.uniones) / 2 $\wedge$ m.#sendas $\leq$ long(m.sendas) $\wedge_{\text{L}}$	2. 5.
($\forall$ e1, e2: string)(e1 $\in$ claves(m.uniones) $\wedge_{\text{L}}$ e2 $\in$ claves(obtener(e1, m.uniones)) $\Rightarrow_{\text{L}}$	
e2 $\in$ claves(m.uniones) $\wedge_{\text{L}}$ e1 $\in$ claves(obtener(e2, m.uniones)) $\wedge_{\text{L}}$	
obtener(e2, obtener(e1, m.uniones)) = obtener(e1, obtener(e2, m.uniones)) $\wedge$	3. 4.
obtener(e2, obtener(e1, m.uniones)) < m.#sendas) $\wedge$	
($\forall$ e1, e2, e3, e4: string)((e1 $\in$ claves(m.uniones) $\wedge_{\text{L}}$ e2 $\in$ claves(obtener(e1, m.uniones)) $\wedge$	
e3 $\in$ claves(m.uniones) $\wedge_{\text{L}}$ e4 $\in$ claves(obtener(e3, m.uniones))) $\Rightarrow_{\text{L}}$	
(obtener(e2, obtener(e1, m.uniones)) = obtener(e4, obtener(e3, m.uniones)) $\Longleftrightarrow$	
(e1 = e3 $\wedge$ e2 = e4) $\vee$ (e1 = e4 $\wedge$ e2 = e3))))	3.

#sendasPorDos : conj(α) c $\times$ dicc($\alpha \times$ dicc($\alpha \times \beta$)) d $\longrightarrow$ nat {c $\subset$ claves(d)}

#sendasPorDos(c, d) $\equiv$ **if** $\emptyset$?(c) **then**
0
else
#claves(obtener(dameUno(c),d)) + #sendasPorDos(sinUno(c), d)
fi

Abs : e_mapa m $\longrightarrow$ mapa {Rep(m)}
Abs(m) $=_{\text{obs}}$ p: mapa |
m.estaciones = estaciones(p) $\wedge_{\text{L}}$
($\forall$ e1, e2: string)((e1 $\in$ estaciones(p) $\wedge$ e2 $\in$ estaciones(p)) $\Rightarrow_{\text{L}}$
(conectadas?(e1, e2, p) $\iff$
e1 $\in$ claves(m.uniones) $\wedge$ e2 $\in$ claves(obtener(e2, m.uniones)))) $\wedge_{\text{L}}$
($\forall$ e1, e2: string)((e1 $\in$ estaciones(p) $\wedge$ e2 $\in$ estaciones(p)) $\wedge_{\text{L}}$
conectadas?(e1, e2, p) $\Rightarrow_{\text{L}}$
(restriccion(e1, e2, p) = m.sendas[obtener(e2, obtener(e1, m.uniones))] $\wedge$ nroConexion(e1, e2, m) = obtener(e2, obtener(e1, m.uniones))) $\wedge$ long(restricciones(p)) = m.#sendas $\wedge_{\text{L}}$ ($\forall$ n:nat) (n <m.#sendas $\Rightarrow_{\text{L}}$ m.sendas[n] = ElemDeSecu(restricciones(p), n)))

2.5. Algoritmos

Algoritmos

iComputadoras(**in** r : `red`) $\rightarrow$ *res* : `itConj(Compu)`
1: *res* $\leftarrow$ CrearIt(*r.computadoras*) $\mathcal{O}(1)$

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

iConectadas?(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ *res* : `bool`
1: *res* $\leftarrow$ Definido?(Significado(*r.vecinosEInterfaces*, c_1.*ip*).*interfaces*, c_2.*ip*) $\mathcal{O}(|c_1| + |c_2|)$

Complejidad: $\mathcal{O}(|c_1| + |c_2|)$

iInterfazUsada(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ *res* : `interfaz`
1: *res* $\leftarrow$ Significado(Significado(*r.vecinosEInterfaces*, c_1.*ip*).*interfaces*, c_2.*ip*) $\mathcal{O}(|c_1| + |c_2|)$

Complejidad: $\mathcal{O}(|c_1| + |c_2|)$

iIniciarRed() $\rightarrow$ *res* : `red`
1: *res* $\leftarrow$ tupla(*vecinosEInterfaces*: Vacío(), *deOrigenADestino*: Vacío(), *computadoras*: Vacío()) $\mathcal{O}(1+1+1)$

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

$\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) =$
$3 * \mathcal{O}(1) = \mathcal{O}(1)$

iAgregarComputadora(**in/out** r : `red`, **in** c : `compu`)
1: Agregar(*r.computadoras*, c) $\mathcal{O}(1)$
2: Definir(*r.vecinosEInterfaces*, *c.ip*, tupla(Vacío(), Vacio())) $\mathcal{O}(|c|)$
3: Definir(*r.deOrigenADestino*, *c.ip*, Vacío()) $\mathcal{O}(|c|)$

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

$\mathcal{O}(1) + \mathcal{O}(|c|) + \mathcal{O}(|c|) =$
$2 * \mathcal{O}(|c|) = \mathcal{O}(|c|)$

iConectar(**in/out** r : `red`, **in** c_1 : `compu`, **in** i_1 : `interfaz`, **in** c_2 : `compu`, **in** i_2 : `interfaz`)

```
1: var tupSig1:tupla ← Significado(r.vecinosEInterfaces, c1.ip)
2: Definir(tupSig1.interfaces, c2.ip, i1)                                          O(|c1| + |c2| + 1)
3: Agregar(tupSig1.compusVecinas, c2)                                              O(1)
4: var tupSig2:tupla ← Significado(r.vecinosEInterfaces, c2.ip)
5: Definir(tupSig2.interfaces, c1.ip, i2)                                          O(|c1| + |c2| + 1)
6: Agregar(tupSig2.compusVecinas, c1)                                              O(1)
7: Definir(Significado(r.deOrigenADestino, c1.ip), c2.ip, CalcularCaminosMinimos(r, c1, c2))   O(1)
8: Definir(Significado(r.deOrigenADestino, c2.ip), c1.ip, CalcularCaminosMinimos(r, c2, c1))   O(1)
```

Complejidad: $\mathcal{O}(|e_1| + |e_2|)$

$\mathcal{O}(|e_1| + |e_2|) + \mathcal{O}(|e_1| + |e_2|) + \mathcal{O}(1) + \mathcal{O}(1) =$
$2 * \mathcal{O}(1) + 2 * \mathcal{O}(|e_1| + |e_2|) =$
$2 * \mathcal{O}(|e_1| + |e_2|) = \mathcal{O}(|e_1| + |e_2|)$

iVecinos(**in** r : `red`, **in** c : `compu`) → res : `conj(compu)`

```
1: res ← Significado(r.vecinosEInterfaces, c.ip).compusVecinas
```

Complejidad:

iUsaInterfaz(**in** r : `red`, **in** c : `compu`, **in** i : `interfaz`) → res : `bool`

```
1: var tupVecinos:tupla ← Significado(r.vecinosEInterfaces, c.ip)                  O(1)
2: var itCompusVecinas:itConj(compu) ← CrearIt(tupVecinos.compusVecinas)           O(1)
3: res:bool ← false                                                                O(1)
4: while HaySiguiente(itCompusVecinas) AND ¬res do                                 O(1)
5:     if Significado(tupVecinos.interfaces, Siguiente(itCompusVecinas).ip) == i then   O(1)
6:         res ← true                                                              O(1)
7:     end if
8:     Avanzar(it)                                                                 O(1)
9: end while
```

Complejidad:

iCaminosMinimos(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) → res : `itConj(`α`)`

```
1: res ← CrearIt(Significado(Significado(r.deOrigenADestino, c1.ip), c2.ip))       O(1)
```

Complejidad:

iHayCamino(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) → res : `bool`

```
1: var conjCaminosMinimos ← CaminosMinimos(r, c1, c2)                              O(1)
2: res ← EsVacio?(conjCaminosMinimos)                                              O(1)
```

Complejidad:

```
iCalcularCaminosMinimos(in r : red, in c1 : compu, in c2 : compu) → res : conj(lista)
 1: res ← Vacio()                                                                          O(1)
 2: var conjCaminosImportantes:conj(lista) = Vacio()                                       O(1)
 3: var pacial:lista ← Vacia()                                                             O(1)
 4: AgregarAtras(parcial, c1)                                                              O(1)
 5: conjCaminosImportantes ← CaminosImportantes(r, c1, c2, parcial)                        O(1)
 6: var itCaminosImportantes:itConj ← CrearIt(conjCaminosImportantes)                       O(1)
 7: while HaySiguiente?(itCaminosImportantes) do                                           O(1)
 8:     if EsVacio?(res) ∨ Longitud(DameUno(res)) = Longitud(Siguiente(itCaminosImportantes)) then  O(1)
 9:         Agregar(res, Siguiente(itCaminosImportantes))                                 O(1)
10:     else
11:         if Longitud(DameUno(res)) < Longitud(Siguiente(itCaminosImportantes)) then    O(1)
12:             res ← Vacio()                                                             O(1)
13:             Agregar(res, Siguiente(itCaminosImportantes))                             O(1)
14:         end if
15:     end if
16: end while
```

Complejidad:

```
iCaminosImportantes(in r : red, in c1 : compu, in c2 : compu, in pacial : lista(compu)) → res : conj(lista)
 1: res ← Vacio()                                                                          O(1)
 2: if Pertenece?(Vecinos(r, c1), c2) then                                                 O(1)
 3:     AgregarAtras(pacial, c2)                                                           O(1)
 4:     Agregar(res, parcial)                                                              O(1)
 5: else
 6:     var itVecinos:itConj ← CrearIt(Vecinos(r, c1))                                     O(1)
 7:     while HaySiguiente?(itVecinos) do                                                  O(1)
 8:         if ¬Pertenece?(parcial, Siguiente(itVecinos)) then                             O(1)
 9:             var auxParcial:lista ← parcial                                             O(1)
10:             AgregarAtras(auxParcial, Siguiente(itVecinos))                             O(1)
11:             Unir(res, CaminosImportantes(r, Siguiente(itVecinos), c2, auxParcial))     O(1)
12:         end if
13:         Avanzar(itVecinos)                                                             O(1)
14:     end while
15: end if
```

Complejidad:

3. DCNet

Interfaz

se explica con: DCNet, Iterador Unidireccional(α).

géneros: `dcnet`.

Operaciones básicas de DCNet

Red(**in** d: `dcnet`) $\rightarrow$ res : `red`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ red(d)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la red del dcnet.

CaminoRecorrido(**in** d: `dcnet`, **in** p: `paquete`) $\rightarrow$ res : `secu(compu)`
Pre $\equiv$ {$p \in$ paqueteEnTransito?(d, p)}
Post $\equiv$ {$res =_{\text{obs}}$ caminoRecorrido(d, p))}
Complejidad: $\mathcal{O}(n * log_2(K))$
Descripción: Devuelve una secuencia con las computadoras por las que paso el paquete.

cantidadEnviados(**in** d: `dcnet`, **in** c: `compu`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$c \in$ computadoras(red(d))}
Post $\equiv$ {$res =_{\text{obs}}$ cantidadEnviados(d, c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de paquetes que fueron enviados desde la computadora.

enEspera(**in** d: `dcnet`, **in** c: `compu`) $\rightarrow$ res : `conj(paquete)`
Pre $\equiv$ {$c \in$ computadoras(red(d))}
Post $\equiv$ {$res =_{\text{obs}}$ enEspera(d, c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve los paquetes que se encuentran en ese momento en la computadora.

IniciarDCNet(**in** r: `red`) $\rightarrow$ res : `dcnet`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ iniciarDCNet(r)}
Complejidad: $\mathcal{O}(1)$
Descripción: Inicia un dcnet con la red y sin paquetes.

CrearPaquete(**in** p: `paquete`, **in**/**out** d: `dcnet`)
Pre $\equiv$ {$d_0 \equiv d \wedge \neg$ ((∃ p_1: paquete)(paqueteEnTransito(s, p_1) $\wedge$ id(p_1) = id(p)) $\wedge$ origen(p) $\in$ computadoras(red(d))$\wedge_{\text{L}}$ destino(p) $\in$ computadoras(red(d))$\wedge_{\text{L}}$ hayCamino?(red(d,origen(p),destino(p)) }
Post $\equiv$ {$res =_{\text{obs}}$ iniciarDCNet(r)}
Complejidad: $\mathcal{O}()$
Descripción: Agrega el paquete al dcnet.

AvanzarSegundo(**in**/**out** d: `dcnet`))
Pre $\equiv$ {$d_0 \equiv d$ }
Post $\equiv$ {$d =_{\text{obs}}$ avanzarSegundo(c_0)}
Complejidad: $\mathcal{O}()$
Descripción: El paquete de mayor prioridad de cada computadora avanza a su proxima computadora siendo esta la del camino mas corto.

Operaciones del iterador

crearIt(**in** c: `ciudad`) $\rightarrow$ res : `itRURs`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ CrearItUni(robots(c))}
Complejidad: $\mathcal{O}(1)$

Descripción: Crea el iterador de robots.

ACTUAL(**in** *it* : `itRURs`) → *res* : `rur`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ Actual(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el actual del iterador de robots.

AVANZAR(**in** *it* : `itRURs`) → *res* : `itRURs`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ Avanzar(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Avanza el iterador de robots.

HAYMAS?(**in** *it* : `itRURs`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ HayMas?(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Se fija si hay mas elementos en el iterador de robots.

3.1. Representacion

Representación

`dcnet` **se representa con** `e_dc`

donde `e_dc` es `tupla`(*red*: `red`, *MasEnviante*: `tupla`(*compu*: `compu`, *enviados*: `nat`),
CompusYPaquetes: `DiccString`(*compu*: `compu`, `tupla`(*PorMasPrioritarios*:DiccRapido(
prioridad :nat, *PaquetitosdePrioridad*:conj(paquete)), *PaquetesYCaminos*:DiccRapido
(*paquetes*:paquete, *CaminoRecorrido*:secu(compu)), *Enviados*:nat))
)

3.2. InvRep y Abs

1. El conjunto de estaciones de 'mapa' es igual al conjunto con todas las claves de 'RURenEst'.
2. La longitud de 'RURs' es mayor o igual a '#RURHistoricos'.
3. Todos los elementos de 'RURs' cumplen que su primer componente ('id') corresponde con su posicion en 'RURs'. Su Componente 'e' es una de las estaciones de 'mapa', su componente 'esta?' es true si y solo si hay estaciones tales que su valor asignado en 'uniones' es igual a su indice en 'RURs'. Su Componente 'inf' puede ser mayor a cero solamente si hay algun elemento en 'sendEv' tal que sea false. Cada elemento de 'sendEv' es igual a verificar 'carac' con la estriccion obtenida al buscar el elemento con la misma posicion en la secuencia de restricciones de 'mapa'.
4. Cada valor contenido en la cola del significado de cada estacion de las claves de 'uniones' pertenecen unicamente a la cola asociada a dicha estacion y a ninguna otra de las colas asociadas a otras estaciones. Y cada uno de estos valores es menor a '#RURHistoricos' y mayor o igual a cero. Ademas la componente 'e' del elemento de la posicion igual a cada valor de las colas asociadas a cada estacion, es igual a la estacion asociada a la cola a la que pertenece el valor.

Rep : e_cr $\longrightarrow$ bool

$\text{Rep}(c) \equiv \text{true} \iff \text{claves(c.RURenEst)} = \text{estaciones(c.mapa)} \wedge$ 1
$\#\text{RURHistoricos} \leq \text{Long(c.RURs)} \wedge_{\text{L}} (\forall$ i:Nat, t:<id:Nat, esta?:Bool, e:String, 2
inf:Nat, carac:Conj(Tag), sendEv: ad(Bool)>)
(i<#RURHistoricos $\wedge_{\text{L}}$ ElemDeSecu(c.RURs, i) = t $\Rightarrow_{\text{L}}$ (t.e $\in$ estaciones(c.mapa) 3
$\wedge$ t.id = i $\wedge$ tam(t.sendEv) = long(Restricciones(c.mapa)) $\wedge$
(t.inf >0 $\Rightarrow$ ($\exists$ j:Nat) (j <tam(t.sendEv) $\wedge_{\text{L}} \neg$ (t.sendEv[j]))) $\wedge$
(t.esta? $\Leftrightarrow$ ($\exists$ e1: String) (e1 $\in$ claves(c.RURenEst) $\wedge_{\text{L}}$ estaEnColaP?(obtener(e1, c.RURenEst), t.id)))
$\wedge$ ($\forall$ h : Nat) (h <tam(t.sendEv) $\Rightarrow_{\text{L}}$
t.sendEv[h] = verifica?(t.carac, ElemDeSecu(Restricciones(c.mapa), h))))) $\wedge_{\text{L}}$
($\forall$ e1, e2: String)(e1 $\in$ claves(c.RURenEst) $\wedge$ e2 $\in$ claves(c.RURenEst) $\wedge$ e1 $\neq$ e2 $\Rightarrow_{\text{L}}$ 4
($\forall$ n:Nat)(estaEnColaP?(obtener(e1, c.RURenEst), n) $\Rightarrow \neg$ estaEnColaP?(obtener(e2, c.RURenEst), n)
$\wedge$ n <#RURHistoricos $\wedge_{\text{L}}$ ElemDeSecu(c.RURs, n).e = e1))

estaEnColaP? : ColaPri $\times$ Nat $\longrightarrow$ Bool

estaEnColaP?(cp, n) $\equiv$ **if** vacia?(cp) **then**
false
else
if desencolar(cp) = n **then**
true
else
estaEnColaP?(Eliminar(cp, desencolar(cp)), n)
fi
fi

Abs : e_cr c $\longrightarrow$ ciudad {Rep(c)}
Abs(c) =$_{\text{obs}}$ u: ciudad |
c.#RURHistoricos = ProximoRUR(U) $\wedge$ c.mapa = mapa(u) $\wedge_{\text{L}}$
robots(u) = RURQueEstan(c.RURs) $\wedge_{\text{L}}$
($\forall$ n:Nat) (n $\in$ robots(u) $\Rightarrow_{\text{L}}$ estacion(n,u) = c.RURs[n].e $\wedge$
tags(n,u) = c.RURs[n].carac $\wedge$ #infracciones(n,u) = c.RURs[n].inf)

RURQueEstan : secu(tupla) $\longrightarrow$ Conj(RUR)

tupla es <id:Nat, esta?:Bool, inf:Nat, carac:Conj(tag), sendEv:arreglo dimensionable(bool)>

RURQueEstan(s) $\equiv$ **if** vacia?(s) **then**
$\emptyset$
else
if Π_2(prim(fin(s))) **then**
Π_1(prim(fin(s))) $\cup$ RURQueEstan(fin(s))
else
RURQueEstan(fin(s))
fi
fi

`it` **se representa con** `e_it`

donde `e_it` es `tupla`(i: `nat`, *maxI*: `nat`, *ciudad*: `puntero(ciudad)`)

Rep : e_it $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ it.i $\leq$ it.maxI $\wedge$ maxI = ciudad.#RURHistoricos
Abs : e_it u $\longrightarrow$ itUni(α) {Rep(u)}

Abs(u) $=_{\text{obs}}$ it: itUni(α) | (HayMas?(u) $\wedge_{\text{L}}$ Actual(u) = ciudad.RURs[it.i] $\wedge$ Siguientes(u, $\emptyset$) = VSiguientes(ciudad, it.i++, $\emptyset$) $\vee$ ($\neg$HayMas?(u))

Siguientes : itUniu $\times$ conj(RURs)cr $\longrightarrow$ conj(RURs)

Siguientes(u, cr) $\equiv$ **if** HayMas(u)? **then** Ag(Actual(Avanzar(u)), Siguientes(Avanzar(u), cr)) **else** Ag($\emptyset$, cr) **fi**

VSiguientes : ciudadc $\times$ Nati $\times$ conj(RURs)cr $\longrightarrow$ conj(RURs)

VSiguientes(u, i, cr) $\equiv$ **if** i <c.#RURHistoricos **then** Ag(c.RURs[i], VSiguientes(u, i++, cr))) **else** Ag($\emptyset$, cr) **fi**

3.3. Algoritmos

Algoritmos

iRed(**in** d: `dcnet`) $\rightarrow$ res : `red`

1: $res \leftarrow (d.red)$ $\mathcal{O}(1)$

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

iCaminoRecorrido(**in** d: `dcnet`,**in** p: `paquete`) $\rightarrow$ res : `secu(compu)`

1: var $it \leftarrow$ computadoras(d.red) $\mathcal{O}(1)$
2: **while** HaySiguiente(it) **do** $\mathcal{O}(1)$
3: **if** definido?(p.id,significado(Siguiente(it),CompusYPaquetes.d).PaquetesYCaminos) **then**
4: $res \leftarrow$ significado(p.id,significado(Siguiente(it),CompusYPaquetes.d).PaquetesYCaminos).CaminoRecorrido
5: **end if**
6: Avanzar(it) $\mathcal{O}(1)$
7: **end while**

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

iCantidadEnviados(**in** d: `dcnet`, **in** c: `compu`) $\rightarrow$ res : `nat`

1: $res \leftarrow$ Significado(c,d.CompusYPaquetes).Enviados $\mathcal{O}(|c|)$

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

enEspera(**in** d: `dcnet`, **in** c: `compu`) $\rightarrow$ res : `itPaquete)`

1: $res \leftarrow$ claves(Significado(c , d.CompusYPaquetes).PaquetesYCaminos) $\mathcal{O}(1)$

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

iIniciarDCNet(**in** r : `red`, **in/out** d: `dcnet`)

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

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

iCrearPaquete(**in** p: `rur`, **in/out** d: `dcnet`)

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

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

iAvanzarSegundo(**in/out** d: `dcnet`)

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

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

iCrear(**in** m: `mapa`) $\rightarrow$ res : `ciudad`

1: $res \leftarrow$ tupla($mapa$: m, $RUREnEst$: Vacío(), $RURs$: Vacía(), $\#RURHistoricos$: 0) $\mathcal{O}(1)$
2: var it:itConj(Estacion) $\leftarrow$ Estaciones(m) $\mathcal{O}(1)$
3: **while** HaySiguiente(it) **do** $\mathcal{O}(1)$
4: Definir($res.RUREnEst$, Siguiente(it), Vacío()) $\mathcal{O}(|e_m|)$
5: Avanzar(it) $\mathcal{O}(1)$
6: **end while**

Complejidad: $\mathcal{O}(Cardinal(Estaciones(m)) * |e_m|)$

$\mathcal{O}(1) + \mathcal{O}(1) + \sum_{i=1}^{Cardinal(Estaciones(m))}(\mathcal{O}(|e_m|) + \mathcal{O}(1)) =$
$2 * \mathcal{O}(1) + Cardinal(Estaciones(m)) * (\mathcal{O}(|e_m|) + \mathcal{O}(1)) =$
$Cardinal(Estaciones(m)) * (\mathcal{O}(|e_m|))$

iEntrar(**in** ts: `conj(tags)`, **in** e: `string`, **in/out** c: `ciudad`)

1: Agregar(Significado($c.RUREnEst$, e), 0, $c.\#RURHistoricos$) $\mathcal{O}(log_2 n + |e|)$
2: Agregar($c.RURs$, $c.\#RURHistoricos$, tupla(id: $c.\#RURHistoricos$, $esta?$: $true$, $estacion$: e, inf: 0, $carac$: ts, $sendEv$: EvaluarSendas(ts, $c.mapa$)) $\mathcal{O}(1 + S * R)$
3: $c.\#RURHistoricos++$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(log_2 n + |e| + S * R)$

$\mathcal{O}(log_2 n + |e|) + \mathcal{O}(1 + S * R) + \mathcal{O}(1) = \mathcal{O}(log_2 n + |e| + S * R)$

iMover(**in** u: `rur`, **in** e: `estación`, **in/out** c: `ciudad`)

1: Eliminar(Significado($c.RUREnEst$, $c.RURs[u].estacion$), $c.RURs[u].inf$, u) $\mathcal{O}(|e| + log_2 N_{e0})$
2: Agregar(Significado($c.RUREnEst$, e), $c.RURs[u].inf$, u) $\mathcal{O}(|e| + log_2 N_e)$
3: **if** $\neg(c.RURs[u].sendEv$[NroConexion($c.RURs[u].estacion$, e, $c.mapa$)]) **then** $\mathcal{O}(|e_0| + |e|)$
4: $c.RURs[u].inf++$ $\mathcal{O}(1)$
5: **end if**
6: $c.RURs[u].estacion \leftarrow e$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(|e| + log_2 N_e)$

$\mathcal{O}(|e| + log_2 N_{e_0}) + \mathcal{O}(|e| + log_2 N_e) + \mathcal{O}(|e_0|, |e|) + max(\mathcal{O}(1), \mathcal{O}(0)) + \mathcal{O}(1) =$
$\mathcal{O}(2 * |e| + log_2 N_e + log_2 N_{e0}) + \mathcal{O}(|e_0| + |e|) + 2 * \mathcal{O}(1) =$
$\mathcal{O}(|e| + log_2 N_e + log_2 N_{e0}) + \mathcal{O}(|e_0| + |e|) =$
$\mathcal{O}(2 * |e| + |e_0| + log_2 N_e + log_2 N_{e0}) = \mathcal{O}(|e| + |e_0| + log_2 N_e + log_2 N_{e0})$ Donde e_0 es c.RURs[u]estacion antes de modificar el valor

iInspección(**in** e: `estación`, **in/out** c: `ciudad`)

1: var rur: nat $\leftarrow$ Desencolar(Significado($c.RUREnEst$, e)) $\mathcal{O}(log_2 N)$
2: $c.RURs[rur].esta? \leftarrow false$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(log_2 N)$

$\mathcal{O}(log_2 N) + \mathcal{O}(1) = \mathcal{O}(log_2 N)$

ICREARIT(**in** c: `ciudad`) $\rightarrow$ res : `itRURs`

1: $itRURS \leftarrow$ tupla$(i : 0, maxI : c.\#RURHistoricos, ciudad : \&c)$ $\mathcal{O}(1)$

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

IACTUAL(**in** it: `itRURs`) $\rightarrow$ res : `rur`

1: $res \leftarrow (it.ciudad \rightarrow RURs)[it.i]$ $\mathcal{O}(1)$

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

IAVANZAR(**in** it: `itRURs`) $\rightarrow$ res : `itRURs`

1: $it.i++$ $\mathcal{O}(1)$

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

IHAYMAS?(**in** it: `itRURs`) $\rightarrow$ res : `bool`

1: $res \leftarrow (it.i < it.maxI)$ $\mathcal{O}(1)$

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

4. Diccionario String(α)

Interfaz

parámetros formales
géneros
función COPIA(**in** d: 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: DICCIONARIO(STRING, α).

géneros: `diccString(`α`)`.

Operaciones básicas de Restricción

VACÍO() $\rightarrow$ res : `diccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacio()}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea nuevo diccionario vacio.

DEFINIR(**in/out** d: `diccString(`α`)`, **in** clv: `string`, **in** def : α)
Pre $\equiv$ {$d_0 =_{\text{obs}} d$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(clv, def, d_0)}
Complejidad: $\mathcal{O}(|clv|)$
Descripción: Agrega un nueva definicion.

DEF?(**in** d: `diccString(`α`)`, **in** clv: `string`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(clv, d)}
Complejidad: $\mathcal{O}(|clv|)$
Descripción: Revisa si la clave ingresada se encuentra definida en el Diccionario.

OBTENER(**in** d: `diccString(`α`)`, **in** clv: `string`) $\rightarrow$ res : `diccString(`α`)`
Pre $\equiv$ {def?(d, clv)}
Post $\equiv$ {$res =_{\text{obs}}$ obtener(clv, d)}
Complejidad: $\mathcal{O}(|clv|)$
Descripción: Devuelve la definicion correspondiente a la clave.

5. DiccRapido

Interfaz

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO).

géneros: `diccRapido`.

Operaciones básicas de DICCRAPIDO

DEF?(**in** c: `clave`, **in** d: `diccRapido`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{def?}(c, d)\}$
Complejidad: $\mathcal{O}(log_2\ n)$, siendo n la cantidad de claves
Descripción: Verifica si una clave está definida.

OBTENER(**in** c: `clave`, **in** d: `diccRapido`) $\rightarrow$ res : `significado`
Pre $\equiv$ $\{\text{def?}(c, d)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{obtener}(c, d)\}$
Complejidad: $\mathcal{O}(log_2\ n)$, siendo n la cantidad de claves
Descripción: Devuelve el significado asociado a una clave

VACÍO() $\rightarrow$ res : `diccRapido`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{vacío}()\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un nuevo diccionario vacío

DEFINIR(**in** c: `clave`, **in** s: `significado`, **in/out** d: `diccRapido`)
Pre $\equiv$ $\{d =_{\text{obs}} d_0\}$
Post $\equiv$ $\{d =_{\text{obs}} \text{definir}(c, s, d_0)\}$
Complejidad: $\mathcal{O}(log_2\ n)$, siendo n la cantidad de claves
Descripción: Define la clave, asociando su significado, al diccionario

BORRAR(**in** c: `clave`, **in/out** d: `diccRapido`)
Pre $\equiv$ $\{d =_{\text{obs}} d_0 \wedge \text{def?}(c, d_0)\}$
Post $\equiv$ $\{d =_{\text{obs}} \text{borrar}(c, d_0)\}$
Complejidad: $\mathcal{O}(log_2\ n)$, siendo n la cantidad de claves
Descripción: Borra la clave del diccionario

CLAVES(**in** d: `diccRapido`) $\rightarrow$ res : `itPaquete`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{claves}(d)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un iterador de paquete

5.1. Representacion

Representación

Para representar el diccionario, elegimos hacerlo sobre AVL. Sabiendo que la cantidad de claves no está acotada, este AVL estará representado con nodos y punteros. Cabe destacar, que las claves del diccionario deben contener una relación de orden. Las claves y los significados se pasan por referencia.

`diccRapido` **se representa con** `estr`

donde `estr` es `tupla`(*raiz*: `puntero`(nodo), *tam*: `nat`)
donde `nodo` es `tupla`(*clave*: `clave`, *significado*: `significado`, *padre*: `puntero`(nodo), *izq*: `puntero`(nodo), *der*: `puntero`(nodo), *alt*: `nat`)

5.2. InvRep y Abs

5.3. Algoritmos

Algoritmos

IDef?(**in** c: `clave`, **in** d: `diccRapido`) $\rightarrow$ res : `bool`

```
1:  var pNodo: puntero(nodo) ← d.raiz                                  O(1)
2:  while *(pNodo) != NULL do                                          O(log2 n)
3:      if *(pNodo).clave == c then                                    O(1)
4:          res ← true                                                 O(1)
5:          return res                                                 O(1)
6:      else
7:          if c > *(pNodo).clave then                                 O(1)
8:              pNodo ← *(pNodo).der                                   O(1)
9:          else
10:             pNodo ← *(pNodo).izq                                   O(1)
11:         end if
12:     end if
13: end while
14: res ← false                                                        O(1)
```

Complejidad: $\mathcal{O}(log_2\ n)$

IObtener(**in** c: `clave`, **in** d: `diccRapido`) $\rightarrow$ res : `significado`

```
1:  var pNodo: puntero(nodo) ← d.raiz                                  O(1)
2:  while *(pNodo).clave != c do                                       O(log2 n)
3:      if c > *(pNodo).clave then                                     O(1)
4:          pNodo ← *(pNodo).der                                       O(1)
5:      else
6:          pNodo ← *(pNodo).izq                                       O(1)
7:      end if
8:  end while
9:  res ← *(pNodo).significado                                         O(1)
```

Complejidad: $\mathcal{O}(log_2\ n)$

IVacío() $\rightarrow$ res : `diccRapido`

```
1:  var res: diccRapido ← tupla(NULL, 0)                               O(1)
```

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

IDefinir(**in** c: `clave`, **in** s: `significado`, **in/out** d: `diccRapido`)

```
1:
2:  if d.raiz == NULL then                                             O(1)
3:      d.raiz ← &tupla(c, s, NULL, NULL, NULL, 1)                     O(1)
4:      d.tam ← 1                                                      O(1)
5:  else
6:      if Def?(c, d) then                                             O(log2 n)
7:          var pNodo: puntero(nodo) ← d.raiz                          O(1)
8:          while *(pNodo).clave != c do                               O(log2 n)
9:              if c > *(pNodo).clave then                             O(1)
10:                 pNodo ← *(pNodo).der                               O(1)
```

```
11:             else
12:                 pNodo ← *(pNodo).izq                                                    O(1)
13:             end if
14:         end while
15:         *(pNodo).significado ← s                                                        O(1)
16:     else
17:         var seguir: bool ← true                                                         O(1)
18:         var pNodo: puntero(nodo) ← d.raiz                                               O(1)
19:         var camino: arreglo[⌊log2 (d.tam)⌋ + 1] de puntero(nodo)                        O(⌊log2 (d.tam)⌋ + 1)
20:         var nroCamino: nat ← 0                                                          O(1)
21:         camino[nroCamino] ← pNodo                                                       O(1)
22:         while seguir == true do                                                         O(log2 n)
23:             if c > *(pNodo).clave ∧ *(pNodo).der == NULL then                           O(1)
24:                 if *(pNodo).izq == NULL then                                            O(1)
25:                     *(pNodo).alt ← 2                                                    O(1)
26:                 else
27:                 end if
28:                 *(pNodo).der ← &tupla(c, s, pNodo, NULL, NULL, 1)                       O(1)
29:                 nroCamino ← nroCamino + 1                                               O(1)
30:                 camino[nroCamino] ← *(pNodo).der                                        O(1)
31:                 seguir ← false                                                          O(1)
32:             else
33:                 if c > *(pNodo).clave ∧ *(pNodo).der != NULL then                       O(1)
34:                     if *(pNodo).izq == NULL then                                        O(1)
35:                         *(pNodo).alt ← *(pNodo).alt + 1                                 O(1)
36:                     else
37:                         *(pNodo).alt ← max(*(*(pNodo).izq).alt, *(*(pNodo).der).alt + 1)   O(1)
38:                     end if
39:                     pNodo ← *(pNodo).der                                                O(1)
40:                     nroCamino ← nroCamino + 1                                           O(1)
41:                     camino[nroCamino] ← pNodo                                           O(1)
42:                 else
43:                     if c < *(pNodo).clave ∧ *(pNodo).izq == NULL then                   O(1)
44:                         if *(pNodo).der ==NULL then                                     O(1)
45:                             *(pNodo).alt ← 2                                            O(1)
46:                         else
47:                         end if
48:                         *(pNodo).izq ← &tupla(c, s, pNodo, NULL, NULL, 1)               O(1)
49:                         nroCamino ← nroCamino + 1                                       O(1)
50:                         camino[nroCamino] ← *(pNodo).izq                                O(1)
51:                         seguir ← false                                                  O(1)
52:                     else
53:                         if *(pNodo).der == NULL then                                    O(1)
54:                             *(pNodo).alt ← *(pNodo).alt + 1                             O(1)
55:                         else
56:                             *(pNodo).alt ← max(*(*(pNodo).izq).alt + 1, *(*(pNodo).der).alt)   O(1)
57:                         end if
58:                         pNodo ← *(pNodo).izq                                            O(1)
59:                         nroCamino ← nroCamino + 1                                       O(1)
60:                         camino[nroCamino] ← pNodo                                       O(1)
61:                     end if
62:                 end if
63:             end if
64:         end while
65:         d.tam ← d.tam + 1                                                               O(1)
66:         while nroCamino ≥ 0 do                                                          O(log2 n)
67:             pNodo ← camino[nroCamino]                                                   O(1)
```

```
68:             if |FactorDesbalance(pNodo)| > 1 then                                  O(1)
69:                 Rotar(pNodo)                                                       O(1)
70:             else
71:             end if
72:             nroCamino ← nroCamino - 1                                              O(1)
73:         end while
74:     end if
75: end if
```

Complejidad: $\mathcal{O}(log_2\ n)$

```
IBorrar(in c: clave, in/out d: diccRapido)
 1: var pNodo: puntero(nodo) ← d.raiz                                                  O(1)
 2: var camino: arreglo[⌊log_2 (d.tam)⌋ + 1] de puntero(nodo)                          O(⌊log_2 (d.tam)⌋ + 1)
 3: var nroCamino: nat ← 0                                                             O(1)
 4: camino[nroCamino] ← pNodo                                                          O(1)
 5: while c != *(pNodo).clave do                                                       O(log_2 n)
 6:     if c > *(pNodo).clave then                                                     O(1)
 7:         if *(pNodo).izq == NULL then                                               O(1)
 8:             *(pNodo).alt ← *(pNodo).alt - 1                                        O(1)
 9:         else
10:             *(pNodo).alt ← max(*(*(pNodo).izq).alt, *(*(pNodo).der).alt - 1)       O(1)
11:         end if
12:         pNodo ← *(pNodo).der                                                       O(1)
13:         nroCamino ← nroCamino + 1                                                  O(1)
14:         camino[nroCamino] ← pNodo                                                  O(1)
15:     else
16:         if *(pNodo).der == NULL then                                               O(1)
17:             *(pNodo).alt ← *(pNodo).alt - 1                                        O(1)
18:         else
19:             *(pNodo).alt ← max(*(*(pNodo).izq).alt - 1, *(*(pNodo).der).alt)       O(1)
20:         end if
21:         pNodo ← *(pNodo).izq                                                       O(1)
22:         nroCamino ← nroCamino + 1                                                  O(1)
23:         camino[nroCamino] ← pNodo                                                  O(1)
24:     end if
25: end while
26: if *(pNodo).izq == NULL ∧ *(pNodo).der == NULL then                               O(1)
27:     if *(pNodo).padre == NULL then                                                 O(1)
28:         d.raiz ← NULL                                                              O(1)
29:         delete pNodo                                                               O(1)
30:     else
31:         if *(pNodo).clave == *(*(*(pNodo).padre).izq).clave then                   O(1)
32:             *(*(pNodo).padre).izq ← NULL                                           O(1)
33:         else
34:             *(*(pNodo).padre).der ← NULL                                           O(1)
35:         end if
36:         delete pNodo  O(1)
37:         nroCamino ← nroCamino - 1                                                  O(1)
38:         while nroCamino ≥ 0 do                                                     O(log_2 n)
39:             pNodo ← camino[nroCamino]                                              O(1)
40:             if |FactorDesbalance(pNodo)| > 1 then                                  O(1)
41:                 Rotar(pNodo)                                                       O(1)
42:             else
43:             end if
44:             nroCamino ← nroCamino - 1                                              O(1)
```

```
45:         end while
46:     end if
47: else
48:     if *(pNodo).izq == NULL ∧ *(pNodo).der != NULL then                    O(1)
49:         if *(pNodo).padre == NULL then                                     O(1)
50:             *(*(pNodo).der).padre ← NULL                                   O(1)
51:             d.raiz ← *(pNodo).der                                          O(1)
52:             delete pNodo                                                   O(1)
53:         else
54:             if *(pNodo).clave == *(*(*(pNodo).padre).izq).clave then       O(1)
55:                 *(*(pNodo).padre).izq ← *(pNodo).der                       O(1)
56:             else
57:                 *(*(pNodo).padre).der ← *(pNodo).der                       O(1)
58:             end if
59:             *(*(pNodo).der).padre ← *(pNodo).padre                         O(1)
60:             delete pNodo                                                   O(1)
61:             nroCamino ← nroCamino - 1                                      O(1)
62:             while nroCamino ≥ 0 do                                         O(log_2 n)
63:                 pNodo ← camino[nroCamino]                                  O(1)
64:                 if |FactorDesbalance(pNodo)| > 1 then                      O(1)
65:                     Rotar(pNodo)                                           O(1)
66:                 else
67:                 end if
68:                 nroCamino ← nroCamino - 1                                  O(1)
69:             end while
70:         end if
71:     else
72:         if *(pNodo).izq != NULL ∧ *(pNodo).der == NULL then
73:             if *(pNodo).padre == NULL then                                 O(1)
74:                 *(*(pNodo).izq).padre ← NULL                               O(1)
75:                 d.raiz ← *(pNodo).izq                                      O(1)
76:                 delete pNodo                                               O(1)
77:             else
78:                 if *(pNodo).clave == *(*(*(pNodo).padre).izq).clave then   O(1)
79:                     *(*(pNodo).padre).izq ← *(pNodo).izq                   O(1)
80:                 else
81:                     *(*(pNodo).padre).der ← *(pNodo).izq                   O(1)
82:                 end if
83:                 *(*(pNodo).izq).padre ← *(pNodo).padre                     O(1)
84:                 delete pNodo                                               O(1)
85:                 nroCamino ← nroCamino - 1                                  O(1)
86:                 while nroCamino ≥ 0 do                                     O(log_2 n)
87:                     pNodo ← camino[nroCamino]                              O(1)
88:                     if |FactorDesbalance(pNodo)| > 1 then                  O(1)
89:                         Rotar(pNodo)                                       O(1)
90:                     else
91:                     end if
92:                     nroCamino ← nroCamino - 1                              O(1)
93:                 end while
94:             end if
95:         else
96:         end if
97:     end if
98: end if
```

Complejidad: $\mathcal{O}(log_2\ n)$

IRotar(**in/out** p: `puntero(nodo)`)

```
1: if FactorDesbalance(p) < 1 then                                  O(1)
2:     if FactorDesbalance(*(p).der) > 1 then                       O(1)
3:         res ← RotarDobleIzq(p)                                   O(1)
4:     else
5:         res ← RotarSimpleIzq(p)                                  O(1)
6:     end if
7: else
8:     if FactorDesbalance(*(p).izq) < 1 then                       O(1)
9:         res ←RotarDobleDer(p)                                    O(1)
10:    else
11:        res ← RotarSimpleDer(p)                                  O(1)
12:    end if
13: end if
```

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

IRotarSimpleIzq(**in/out** p: `puntero(nodo)`)

```
1: var r: puntero(nodo) ← p                                         O(1)
2: var r2: puntero(nodo) ← *(r).der                                 O(1)
3: var i: puntero(nodo) ← *(r).izq                                  O(1)
4: var i2: puntero(nodo) ← *(r2).izq                                O(1)
5: var d2: puntero(nodo) ← *(r2).der                                O(1)
6: var padre: puntero(nodo) ← *(r).padre                            O(1)
7: if padre != NULL then
8:     if *(r).clave == *(*(padre).izq).clave then                  O(1)
9:         *(padre).izq ← r2                                        O(1)
10:    else
11:        *(padre).der ← r2                                        O(1)
12:    end if
13: else
14: end if
15: *(r2).padre ← padre                                             O(1)
16: *(r2).izq ← r                                                   O(1)
17: *(r).padre ← r2                                                 O(1)
18: *(r).der ← i2                                                   O(1)
19: if i2 != NULL then
20:    *(i2).padre ← r                                              O(1)
21: else
22: end if
23: *(r).alt ← Altura(r)                                            O(1)
24: *(r2).alt ← Altura(r2)                                          O(1)
```

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

IRotarSimpleDer(**in/out** p: `puntero(nodo)`)

```
1: var r: puntero(nodo) ← p                                         O(1)
2: var r2: puntero(nodo) ← *(r).izq                                 O(1)
3: var d: puntero(nodo) ← *(r).der                                  O(1)
4: var i2: puntero(nodo) ← *(r2).izq                                O(1)
5: var d2: puntero(nodo) ← *(r2).der                                O(1)
6: var padre: puntero(nodo) ← *(r).padre                            O(1)
7: if padre != NULL then
8:     if *(r).clave == *(*(padre).izq).clave then                  O(1)
9:         *(padre).izq ← r2                                        O(1)
```

```
10:     else
11:         *(padre).der ← r2                                              O(1)
12:     end if
13: else
14: end if
15: *(r2).padre ← padre                                                    O(1)
16: *(r2).der ← r                                                          O(1)
17: *(r).padre ← r2                                                        O(1)
18: *(r).izq ← d2                                                          O(1)
19: if d2 != NULL then
20:     *(d2).padre ← r                                                    O(1)
21: else
22: end if
23: *(r).alt ← Altura(r)                                                   O(1)
24: *(r2).alt ← Altura(r2)                                                 O(1)
```

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

IRotarDobleIzq(**in**/**out** p: `puntero(nodo)`, **in**/**out** d: `diccRapido`)

```
1: RotarSimpleDer(*(p).der)                                                O(1)
2: RotarSimpleIzq(p)                                                       O(1)
```

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

IRotarDobleDer(**in**/**out** p: `puntero(nodo)`)

```
1: RotarSimpleIzq(*(p).izq)                                                O(1)
2: RotarSimpleDer(p)                                                       O(1)
```

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

IAltura(**in** p: `puntero(nodo)`) → res : `nat`

```
1: if *(p).izq == NULL ∧ *(p).der == NULL then                             O(1)
2:     res ← 1                                                             O(1)
3: else
4:     if *(p).izq != NULL ∧ *(p).der == NULL then                         O(1)
5:         res ← *(*(p).izq).alt + 1                                       O(1)
6:     else
7:         if *(p).izq == NULL ∧ *(p).der != NULL then                     O(2)
8:             res ← *(*(p).der).alt + 1                                   O(1)
9:         else
10:            res ← max(*(*(p).izq).alt, *(*(p).der).alt) + 1             O(1)
11:        end if
12:    end if
13: end if
```

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

IFactorDesbalance(**in** p: `puntero(nodo)`) → res : `nat`

```
1: if *(p).izq == NULL ∧ *(p).der == NULL then                             O(1)
2:     res ← 0                                                             O(1)
3: else
4:     if *(p).izq != NULL ∧ *(p).der == NULL then                         O(1)
5:         res ← *(*(p).izq).alt                                           O(1)
```

```
6:     else
7:         if *(p).izq == NULL ∧ *(p).der != NULL then                    O(2)
8:             res ← - *(*(p).der).alt                                    O(1)
9:         else
10:            res ← *(*(p).izq).alt - *(*(p).der).alt                    O(1)
11:        end if
12:    end if
13: end if
```

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

6. Extensión de Lista Enlazada(α)

6.1. Interfaz

Interfaz

se explica con: SECU(α), ITERADOR BIDIRECCIONAL(α).

géneros: `lista, itLista(`α`)`.

Operaciones básicas de lista

PERTENECE?(**in** l: `lista`, **in** e: α) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ está?(l, e)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve true o false según si el elemento pertenece o no a la lista

6.2. Algoritmos

Algoritmos

PERTENECE?(**in** l: `lista(`α`)`, **in** e: `lista`) $\rightarrow$ res : `bool`

```
1: var itLista ← CrearIt(l)                                   O(1)
2: res ← false                                                O(1)
3: while HaySiguiente(itLista) AND ¬res do                    O(1)
4:     if Siguiente(itLista) == e then                        O(1)
5:         res ← true                                         O(1)
6:     end if
7:     Avanzar(itLista)                                       O(1)
8: end while
```

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

7. Extensión de Conjunto Lineal(α)

7.1. Interfaz

Interfaz

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

géneros: `conj`, `itConj`(α).

Operaciones básicas de Conjunto

UNION(**in**/**out** c: `conj`(α), **in** d: `conj`(α)) $\rightarrow$ res : `itConj`(α)
Pre $\equiv \{c =_{\text{obs}} c_0\}$
Post $\equiv \{res =_{\text{obs}} crearItBi(c \cup d)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Modifica el c para que contenga la unión de los dos conjuntos pasados como parámetro
Aliasing: Los elementos de c se copian a d

DAMEUNO(**in** c: `conj`(α)) $\rightarrow$ res : α
Pre $\equiv \{\#(c) > 0\}$
Post $\equiv \{res =_{\text{obs}} DameUno(c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un elemento cualquiera del conjunto

7.2. Algoritmos

Algoritmos

UNION(**in**/**out** c: `conj`(α), **in** d: `conj`(α)) $\rightarrow$ res : `itConj`(α)

```
1: var itConj ← CrearIt(d)                                   O(1)
2: while HaySiguiente(itConj) do                             O(1)
3:     Agregar(c, Siguiente(itConj))                         O(1)
4:     Avanzar(itConj)                                       O(1)
5: end while
6: var res ← CrearIt(c)                                      O(1)
```

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

DAMEUNO(**in** c: `conj`(α))

```
1: var itConj ← CrearIt(c)                                   O(1)
2: res ← Siguiente(itConj))                                  O(1)
```

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