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)

1.3. Diccionario(Clave, Significado)

otras operaciones
vacío? : Diccionario $\longrightarrow$ Bool
claveMax : Diccionario d $\longrightarrow$ Clave $\{\neg\text{vacío(d)}\}$
secuClaves: Diccionario $\longrightarrow$ Secu(clave)

axiomas
vacío?(vacío) $\equiv$ true
vacío?(definir(c, s, d)) $\equiv$ false
claveMax(d) $\equiv$ elemMax(claves(d))
secuClaves(vacío) $\equiv$ <>
secuClaves(definir(c, s, d)) $\equiv$ secuClaves(d) $\circ$ c

1.4. Conjunto(α<)

otras operaciones
elemMax : Conj(α) c $\longrightarrow$ α $\{\neg\emptyset?(c)\}$
auxElemMax : α × Conj(α) $\longrightarrow$ α

axiomas
elemMax(c) $\equiv$ auxMaxElem(dameUno(c), c)

```
auxElemMax(e, c) ≡ if ∅?(c) then
                       e
                   else
                       if e > dameUno(c) then
                           auxElemMax(e, sinUno(c))
                       else
                           auxElemMax(dameUno(c), sinUno(c))
                       fi
                   fi
```

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`(*directasEInterfaces*: `diccString`(*compu*: `string`, tupla(*directas*: `diccString`(*compu*: `string`, *interfaz*: `nat`), *compusDirectas*: `conj(compu)`))
, *deOrigenADestino*: `diccString`(*compu*: `string`, *indirectas*: `diccString` (*compu*: `string`, *caminosMinimos*: `conj(lista(compu))`))
, *computadoras*: `conj(compu)`)

2.4. InvRep y Abs

1. Las claves de *directasEInterfaces* son las mismas que las de *deOrigenADestino* y también que el conjunto formado por las ip de *computadoras*.

2.

Rep : e_mapa $\longrightarrow$ bool
Rep(m) $\equiv$ true $\iff$
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)) $\iff$
(e1 = e3 $\wedge$ e2 = e4) $\vee$ (e1 = e4 $\wedge$ e2 = e3)))) 3.

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

#sendasPorDos(c, d) ≡
if ∅?(c) **then** 0 **else** #claves(obtener(dameUno(c),d)) + #sendasPorDos(sinUno(c), d) **fi**

Abs : e_mapa m $\longrightarrow$ mapa {Rep(m)}
Abs(m) =$_{obs}$ p: mapa |
m.estaciones = estaciones(p) $\wedge_L$
($\forall$ e1, e2: string)((e1 $\in$ estaciones(p) $\wedge$ e2 $\in$ estaciones(p)) $\Rightarrow_L$
(conectadas?(e1, e2, p) $\iff$
e1 $\in$ claves(m.uniones) $\wedge$ e2 $\in$ claves(obtener(e2, m.uniones)))) $\wedge_L$
($\forall$ e1, e2: string)((e1 $\in$ estaciones(p) $\wedge$ e2 $\in$ estaciones(p)) $\wedge_L$
conectadas?(e1, e2, p) $\Rightarrow_L$
(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 $\wedge_L$ ($\forall$ n:nat) (n <m.#sendas $\Rightarrow_L$ 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($r.directasEInterfaces$, $c_1.ip$).$directas$, $c_2.ip$) $\mathcal{O}(|c_1| + |c_2|)$

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($r.directasEInterfaces$, $c_1.ip$).$directas$, $c_2.ip$) $\mathcal{O}(|c_1| + |c_2|)$

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

iIniciarRed() $\rightarrow$ res : `red`
1: $res \leftarrow$ tupla($directasEInterfaces$: 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($r.directasEInterfaces$, $c.ip$, 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($r.directasEInterfaces$, $c_1.ip$)
2: Definir($tupSig1.directas$, $c_2.ip$, i_1) $\mathcal{O}(|c_1| + |c_2| + 1)$
3: Agregar($tupSig1.compusDirectas$, c_2) $\mathcal{O}(1)$
4: var $tupSig2$:tupla ← Significado($r.directasEInterfaces$, $c_2.ip$)
5: Definir($tupSig2.directas$, $c_1.ip$, i_2) $\mathcal{O}(|c_1| + |c_2| + 1)$
6: Agregar($tupSig2.compusDirectas$, c_1) $\mathcal{O}(1)$
7: Definir(Significado($r.deOrigenADestino$, $c_1.ip$), $c_2.ip$, CalcularCaminosMinimos(r, c_1, c_2)) $\mathcal{O}(1)$
8: Definir(Significado($r.deOrigenADestino$, $c_2.ip$), $c_1.ip$, CalcularCaminosMinimos(r, c_2, c_1)) $\mathcal{O}(1)$

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($r.directasEInterfaces$, $c.ip$).$compusDirectas$

Complejidad:

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

1: var $tupVecinos$:tupla ← Significado($r.directasEInterfaces$, $c.ip$) $\mathcal{O}(1)$
2: var $itcompusDirectas$:itConj(compu) ← CrearIt($tupVecinos.compusDirectas$) $\mathcal{O}(1)$
3: res:bool ← $false$ $\mathcal{O}(1)$
4: **while** HaySiguiente($itcompusDirectas$) AND ¬res **do** $\mathcal{O}(1)$
5: **if** Significado($tupVecinos.directas$, Siguiente($itcompusDirectas$).ip) == i **then** $\mathcal{O}(1)$
6: res ← $true$ $\mathcal{O}(1)$
7: **end if**
8: Avanzar(it) $\mathcal{O}(1)$
9: **end while**

Complejidad:

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

1: res ← CrearIt(Significado(Significado($r.deOrigenADestino$, $c_1.ip$), $c_2.ip$)) $\mathcal{O}(1)$

Complejidad:

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

1: var $conjCaminosMinimos$ ← CaminosMinimos(r, c_1, c_2) $\mathcal{O}(1)$
2: res ← EsVacio?($conjCaminosMinimos$) $\mathcal{O}(1)$

Complejidad:

iCalcularCaminosMinimos(**in** r : red, **in** c_1 : compu, **in** c_2 : compu) → res : conj(lista)

1: res ← Vacio() $\mathcal{O}(1)$
2: var $conjCaminosImportantes$:conj(lista) = Vacio() $\mathcal{O}(1)$
3: var $pacial$:lista ← Vacia() $\mathcal{O}(1)$
4: AgregarAtras($parcial$, c_1) $\mathcal{O}(1)$
5: $conjCaminosImportantes$ ← CaminosImportantes(r, c_1, c_2, $parcial$) $\mathcal{O}(1)$

```
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** c_1 : `compu`, **in** c_2 : `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

3.1. Interfaz

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: $\mathcal{O}(|c.id|)$
Descripción: Devuelve la cantidad de paquetes que fueron enviados desde la computadora.

enEspera(**in** d: `dcnet`, **in** c: `compu`) $\rightarrow$ res : `itPaquete`
Pre $\equiv$ {$c \in$ computadoras(red(d))}
Post $\equiv$ {$res =_{\text{obs}}$ enEspera(d, c)}
Complejidad: $\mathcal{O}(|c.id|)$
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: $\mathcal{O}(N * L)$
Descripción: Inicia un dcnet con la red y sin paquetes.

CrearPaquete(**in** p: `paquete`, **in/out** d: `dcnet`)
Pre $\equiv$ {$d_0 \equiv d \wedge \neg$ (($\exists$ 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: $\mathcal{O}(L + log_2(k))$
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: $\mathcal{O}(N * (L + log_2(k))$
Descripción: El paquete de mayor prioridad de cada computadora avanza a su proxima computadora siendo esta la del camino mas corto.

PaqueteEnTransito?(**in** d: `dcnet`, **in** p: `paquete`) $\rightarrow$ res : `bool`
Pre $\equiv$ { }
Post $\equiv$ {$res =_{\text{obs}}$ paqueteEnTransito?(d,p)}
Complejidad: $\mathcal{O}(N * log_2(k))$
Descripción: Devuelve si el paquete esta o no en alguna computadora del sistema.

LaQueMasEnvio(**in** d: `dcnet`)) $\rightarrow$ res : `compu`
Pre $\equiv$ {}
Post $\equiv$ {$res =_{\text{obs}}$ laQueMasEnvio(d)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la computadora que mas paquetes envio.

Operaciones del iterador

3.2. Representacion

Representación

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

donde `e_dc` es `tupla`(*red*: `red`,
MasEnviante: `tupla`(*compu*: `compu`, *enviados*: `nat`),
CompYPaq: `DiccString`(*compu*: `compu`, `tupla`(*MasPriori*:DiccRapido(*prioridad* :nat, *PaqdePriori*:conj(paquete)), *PaqYCam*:DiccRapido (*paq*:paquete, *CamRecorrido*:secu(compu)
)

3.3. 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
Rep(c) $\equiv$ true $\iff$ claves(c.RURenEst) = estaciones(c.mapa) $\wedge$ 1
#RURHistoricos $\leq$ 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) $=_{obs}$ u: ciudad |
c.#RURHistoricos = ProximoRUR(U) $\wedge$ c.mapa = mapa(u) $\wedge_L$
robots(u) = RURQueEstan(c.RURs) $\wedge_L$
($\forall$ n:Nat) (n $\in$ robots(u) $\Rightarrow_L$ 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) $=_{obs}$ it: itUni(α) | (HayMas?(u) $\wedge_L$ Actual(u) = ciudad.RURs[it.i] $\wedge$ Siguientes(u, $\emptyset$) = VSiguientes(ciudad, it.i++, $\emptyset$) $\vee$ ($\neg$HayMas?(u))

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

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 $\times$ $\longrightarrow$ conj(RURs)
conj(RURs)cr

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

3.4. Algoritmos

Algoritmos

iRed(**in** d: dcnet) → res : red

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

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

iCaminoRecorrido(**in** d: dcnet,**in** p: paquete) → res : secu(compu)

1: var $it \leftarrow$ computadoras(d.red) $\mathcal{O}(1)$
2: var *esta*: bool ← false
3: **while** HaySiguiente(it) ∧ ¬ esta **do** $\mathcal{O}(n)$
4: var:diccRapido *diccpaq* ← obtener(Siguiente(it).id, d.CompYPaq).PaqYCam) $\mathcal{O}(L)$
5: **if** def?(p,*diccpaq*) **then** $\mathcal{O}(L + log_2(N))$
6: *esta* ← true $\mathcal{O}(1)$
7: $res \leftarrow$ obtener(p, *diccpaq*).CamRecorrido $\mathcal{O}(L + log_2(N) + 1)$
8: **end if**
9: Avanzar(it) $\mathcal{O}(1)$
10: **end while**

Complejidad: $\mathcal{O}()$

iCantidadEnviados(**in** d: dcnet, **in** c: compu) → res : nat

1: $res \leftarrow$ obtener(c.id,d.CompYPaq).Enviados $\mathcal{O}(L)$

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

Siendo L la longitud de el ID de c

iEnEspera(**in** d: dcnet, **in** c: compu) → res : itPaquete)

1: $res \leftarrow$ claves(obtener(c.id, d.CompYPaq).PaqYCam) $\mathcal{O}(L)$

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

Siendo L la longitud del ID de c

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

1: d.red ← r $\mathcal{O}(NOSE)$
2: var $it \leftarrow$ computadoras(red) $\mathcal{O}(1)$
3: d.MasEnviante ← tupla(Siguiente(it),0) $\mathcal{O}(1)$
4: d.CompyPaq ← Vacio() $\mathcal{O}(1)$
5: **while** HaySiguiente(it) **do** $\mathcal{O}(N)$
6: Definir(Siguiente(it).id, tupla(Vacio(),Vacio(), 0), d.CompyPaq) $\mathcal{O}(L + 1 + 1)$
7: Avanzar(it) $\mathcal{O}(1)$
8: **end while**

Complejidad: $\mathcal{O}(N * L)$

Siendo N la cantidad de computadoras en la red y L el ID mas largo de ellas.

```
iCrearPaquete(in p: paquete, in/out d: dcnet)
 1: var diccprio: diccRapido ← obtener(p.origen, d.CompYPaq).MasPriori)                          O(L)
 2: var dicccam: diccRapido ← obtener(p.origen, d.CompYPaq).PaqYCam)                             O(L)
 3: if ¬ def?(p.prioridad, diccprio) then                                                        O(log2(s))
 4:     Definir(p.prioridad,Agregar(Vacio(), p), diccprio)                                       O(log2(s))
 5: else
 6:     Definir(p.prioridad,Agregar(obtener(p.prioridad, diccprio), p), diccprio                 O(log2(s))
 7: end if
 8: definir(p, dicccam, AgregarAtras(<>,p.origen)                                                O(log2(k))
```

Complejidad: $\mathcal{O}(L + log_2(k))$

```
iAvanzarSegundo(in/out d: dcnet)
 1: var it ← computadoras(red)                                                                   O(1)
 2: var aux ← Vacia()                                                                            O(1)
 3: while HaySiguiente(it) do                                                                    O(N)
 4:     var diccprio: diccRapido ← obtener( Siguiente(it).id, d.CompYPaq).MasPriori)             O(L)
 5:     var dicccam: diccRapido ← obtener(Siguiente(it).id, d.CompYPaq).PaqYCam)                 O(L)
 6:     if ¬ Vacio?(diccprio) then                                                               O(1)
 7:         var paq: paquete ← Primero(obtener(DameMax(diccprio), diccprio))                     O(log2(k) + 1 + 1)
 8:         AgregarAdelante(aux, tupla(paq: paq,pcant: it.id, camrecorrido: obtener(paq, dicccam))  O(1 + log2(k))
 9:         Eliminar(obtener(DameMax(diccprio), diccprio), paq)                                  O(log2(k) + log2(k) + 1)
10:         if EsVacio?(obtener(DameMax(diccprio), diccprio) then                                O(log2(k))
11:             borrar(DameMax(diccprio), diccprio)                                              O(log2(k))
12:         end if
13:         borrar(paq, dicccam)                                                                 O(log2(k))
14:         obtener(Siguiente(it).id, d.CompYPaq).Enviados ++                                    O(L)
15:         if obtener(Siguiente(it).id, d.CompYPaq).Enviados >(d.MasEnviante).enviados then     O(L + 1)
16:             d.MasEnviante ← tupla(Siguiente(it), obtener(Siguiente(it).id, d.CompYPaq).Enviados )  O(L+ 1)
17:         end if
18:     end if
19:     Avanzar(it)                                                                              O(1)
20: end while
21: var itaux ← crearIt(aux)                                                                     O(1)
22: while HaySiguiente(itaux) do                                                                 O(Nk)
23:     var proxpc: compu ← Primero(Siguiente(CaminosMinimos(d.red, itaux.pcant, itaux.destino))  O(L1+ L2)
24:     var diccprio: diccRapido ← obtener( proxpc.id, d.CompYPaq).MasPriori)                    O(L)
25:     var dicccam: diccRapido ← obtener(proxpc.id, d.CompYPaq).PaqYCam)                        O(L)
26:     if def?((itaux.paq).prioridad, diccprio) then                                            O(log2(k))
27:         var mismaprio: conj(paquetes) ← Agregar(obtener(it3.paq.prioridad, diccprio), it3.paq)  O(log2(k))
28:         definir((it3.paq).prioridad, mismaprio, diccprio)                                    O(log2(k))
29:     else
30:         Definir(it3.prioridad, Agregar(Vacio(),it3.paq), diccprio)                           O(log2(k))
31:     end if
32:     definir(p.paq,AgregarAtras(it3.camrecorrido ,proxpc), dicccam)                           O(log2(k))
33:     EliminarSiguiente(it3)                                                                   O(1)
34:     Avanzar(it3)                                                                             O(1)
35: end while
```

Complejidad: $\mathcal{O}(N * (L + log_2(k))$

iPaqueteEnTransito?(**in** d: `dcnet`, **in** p: `paquete`) $\rightarrow$ res : `bool`

```
1: var it ← crearIt(computadoras(d.red)                                  O(1)
2: var esta: bool ← false                                                O(1)
3: while HaySiguiente(it) ∧ ¬ esta  do                                   O()
4:     esta ← def?(obtener(d.CompYPaq,i.id).PaqYCam , p)                 O(log_2(k))
5:     Avanzar(it)                                                       O(1)
6: end while
7: res ← esta                                                            O(1)
```

Complejidad: $\mathcal{O}(N * log(k))$

iLaQueMasEnvio(**in** d: `dcnet`) $\rightarrow$ res : `compu`

```
1: res ← (d.MasEnviante).compu                                           O(1)
```

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

4. Diccionario String

4.1. Interfaz

Interfaz

se explica con: DICCIONARIO(STRING, α).

géneros: `diccString(`α`)`.

Operaciones básicas de Diccionario String(α)

DEF?(**in** clv: `string`, **in** d: `diccString(`α`)`) $\rightarrow$ 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.

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

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.

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

BORRAR(**in** clv: `string`, **in/out** d: `diccString(`α`)`)
Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge$ def?(clv, d_0)}
Post $\equiv$ {$res =_{\text{obs}}$ borrar(k, d_0)}
Complejidad: $\mathcal{O}(|clv|)$
Descripción: Borra la definición.

5. Diccionario Rápido

5.1. Interfaz

Interfaz

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO).

géneros: `diccRapido(`α`,`β`)`.

Operaciones básicas de Diccionario Rápido(α,β)

DEF?(**in** c: α, **in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ 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: α, **in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : β
Pre $\equiv$ {def?(c, d)}
Post $\equiv$ {$res =_{\text{obs}}$ 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(`α`,`β`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacío()}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un nuevo diccionario vacío

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

VACÍO?(**in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacío?(d)}
Complejidad: $\mathcal{O}(1)$
Descripción: Verifica si el diccionario vacío

CLAVEMAX(**in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : α
Pre $\equiv$ {$\neg$vacío?(d)}
Post $\equiv$ {$res =_{\text{obs}}$ claveMax(d)}
Complejidad: $\mathcal{O}(log_2\ n)$
Descripción: Devuelve la mayor clave

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

Operaciones del Iterador

CrearIt(**in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : `itClave`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearItUni(secuClaves(d))}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea el iterador de claves

HayMas?(**in** it: `itClave`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ HayMas?(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Verifica si hay más elementos a iterar

Actual(**in** it: `itClave`) $\rightarrow$ res : α
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ {res Actual(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el actual del iterador

Avanzar(**in/out** it: `itClave`)
Pre $\equiv$ {$it =_{\text{obs}} it_0 \wedge$ HayMas?(it_0)}
Post $\equiv$ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $\mathcal{O}(n)$
Descripción: Avanza el iterador

5.2. Auxiliares

Operaciones auxiliares

DameNodos(**in** p: `puntero(nodo)`, **in** $actual$: `nat`, **in** $destino$: `nat`) $\rightarrow$ res : `conj(nodo)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ nodosNivel($p, actual, destino$)}
Complejidad: $\mathcal{O}(n)$
Descripción: Crea un conjunto de nodos con todos los nodos pertenecientes al nivel destino

Rotar(**in/out** p: `puntero(nodo)`)
Pre $\equiv$ {$p =_{\text{obs}} p_0 \wedge$ p != NULL $\wedge_{\text{L}}$ (
*(p).der != NULL $\vee$
*(p).izq != NULL $\vee$
(*(p).der !=NULL $\wedge_{\text{L}}$ *(*(p).der).izq != NULL) $\vee$
(*(p).izq != NULL $\wedge_{\text{L}}$ *(*(p).izq).der != NULL))}
Post $\equiv$ {$p =_{\text{obs}}$ rotar(p_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Realiza la rotación pertinente de p, de ser necesario

RotarSimpleIzq(**in/out** p: `puntero(nodo)`)
Pre $\equiv$ {$p =_{\text{obs}} p_0 \wedge$ p != NULL $\wedge_{\text{L}}$ *(p).der != NULL}
Post $\equiv$ {$p =_{\text{obs}}$ rotarSimpleIzq(p_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Realiza una rotación simple izquierda del nodo p, y los nodos involucrados

RotarSimpleDer(**in/out** p: `puntero(nodo)`)
Pre $\equiv$ {$p =_{\text{obs}} p_0 \wedge$ p != NULL $\wedge_{\text{L}}$ *(p).izq != NULL}
Post $\equiv$ {$p =_{\text{obs}}$ rotarSimpleDer(p_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Realiza una rotación simple derecha del nodo p, y los nodos involucrados

RotarDobleIzq(**in/out** p: `puntero(nodo)`)
Pre $\equiv$ {$p =_{\text{obs}} p_0 \wedge$ p != NULL $\wedge_{\text{L}}$ *(p).der !=NULL $\wedge_{\text{L}}$ *(*(p).der).izq != NULL}
Post $\equiv$ {$p =_{\text{obs}}$ rotarDobleIzq(p_0)}

Complejidad: $\mathcal{O}(1)$
Descripción: Realiza una rotación doble izquierda del nodo p, y los nodos involucrados

RotarDobleDer(**in/out** p: `puntero(nodo)`)
Pre $\equiv$ {$p =_{\text{obs}} p_0 \wedge$ p != NULL $\wedge_{\text{L}}$ *(p).izq != NULL $\wedge_{\text{L}}$ *(*(p).izq).der != NULL}
Post $\equiv$ {$p =_{\text{obs}}$ rotarDobleDer(p_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Realiza una rotación doble derecha del nodo p, y los nodos involucrados

Altura(**in** p: `puntero(nodo)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {p != NULL}
Post $\equiv$ {$res =_{\text{obs}}$ altura(p)}
Complejidad: $\mathcal{O}(1)$
Descripción: Calcula y devuelve la altura actual de p

FactorDesbalance(**in** p: `puntero(nodo)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {p != NULL}
Post $\equiv$ {$res =_{\text{obs}}$ factorDesbalance(p)}
Complejidad: $\mathcal{O}(1)$
Descripción: Calcula y devuelve el factor de desbalance actual de p

5.3. Representación

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(`α`,`β`)` **se representa con** `estr`

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

5.4. InvRep y Abs

InvRep en lenguaje coloquial:

1. La componente "tam" es igual a la cantidad de nodos del árbol.
2. Todo nodo del árbol tiene padre, con excepción de la raíz, que no tiene padre. Y de tener padre, como máximo, puede existir otro nodo que tenga el mismo padre.
3. No hay dos nodos con el mismo hijo izquierdo, ni hay dos nodos con el mismo hijo derecho.
4. Un nodo (n1) tiene a otro nodo (n2) como hijo (ya sea izquierdo, o derecho), si y solo si n2 tiene a n1 como padre.
5. Un nodo no puede tener al mismo hijo izquierdo y derecho. Tampoco puede tenerse a sí mismo como padre, o hijo izquierdo, o derecho.
6. La relación de orden es total.
7. Un nodo es mayor a otro si la componente "clave" del primero es mayor que la del segundo.
8. Un nodo es menor a otro si la componente "clave" del primero es menor que la del segundo.
9. No hay dos nodos con la misma componente "clave".
10. Para todo nodo, todos los nodos de su subárbol derecho son mayores a él.
11. Para todo nodo, todos los nodos del su subárbol izquierdo son menores que él.

12. La componente "alt" de cada nodo es igual a la cantidad de nodos que hay que "bajar" para llegar a su hoja mas lejana + 1. Vale aclarar que el nodo hoja tiene la componente "alt" igual a 1.

13. Para todo nodo, la diferencia, en módulo, de la altura entre sus subárboles es menor o igual a 1.

Abs:

Abs : estr e $\longrightarrow$ Diccionario(Clave, Significado) {Rep(e)}
Abs(e) $=_{\text{obs}}$ d: Diccionario(Clave, Significado) |
(∀c: clave)
def?(c, d) = Def?(c, e) $\wedge_{\text{L}}$
obtener(c, d) = Obtener(c, e)

5.5. Algoritmos

Algoritmos

IDEF?(**in** c: α, **in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ 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: $\mathcal{O}(log_2\ n)$

Siendo n la cantidad de nodos.
Vamos a ignorar los condicionales y las asignaciones, dado que éstas siempre ocurren en tiempo constante. Para analizar la complejidad del ciclo, es necesario tomar en cuenta cuantas iteraciones (como máximo) haría éste antes de romper su guarda. Como se trata de buscar un nodo en un AVL, sabemos que la búsqueda es $log_2\ n$, dado que el árbol está balanceado, es decir, en el peor caso estaremos buscando un nodo que puede pertenecer (o no) al último nivel y por esto se debe descender (como máximo) $log_2\ n$ veces. Luego, la complejidad del ciclo es la que define la complejidad del algoritmo.

IOBTENER(**in** c: α, **in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : β

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

Siendo n la cantidad de nodos.
Es un algoritmo muy parecido al de DEF?. Nuevamente ignoraremos los condicionales y las asignaciones, dado

que éstas siempre ocurren en tiempo constante. Para analizar la complejidad del ciclo, es necesario tomar en cuenta cuantas iteraciones (como máximo) haría éste antes de romper su guarda. Como se trata de buscar un nodo en un AVL, sabemos que la búsqueda es $log_2\ n$, dado que el árbol está balanceado, es decir, en el peor caso estaremos buscando un nodo que puede pertenecer (o no) al último nivel y por esto se debe descender (como máximo) $log_2\ n$ veces. Luego, la complejidad del ciclo es la que define la complejidad del algoritmo.

IVacío() $\rightarrow$ *res* : `diccRapido(`α`,`β`)`

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

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

IDefinir(**in** c: α, **in** s: β, **in/out** d: `diccRapido(`α`,`β`)`)

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

Siendo n la cantidad de nodos.
En este algoritmo, tomaremos en cuenta las complejidades de tres casos e ignoraremos los condicionales y asignaciones (dado que son constantes).
-El primer caso es cuando se quiera definir en un diccionario vacío, esto es $\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) = 3 * \mathcal{O}(1) = \mathcal{O}(1)$
-El segundo caso es cuando se quiera definir una clave que ya estaba definida previamente, aquí ignoraremos las asignaciones y condicionales (cuyas complejidades son $\mathcal{O}(1)$), y nos centraremos en el uso de Def? y el ciclo. Def? sabemos que toma tiempo logarítmico, y en cuanto al ciclo, sabemos que tomará tiempo logarítmico también, porque iterará hasta buscar el nodo buscado. Esto es: $\mathcal{O}(log_2\ n) + \mathcal{O}(log_2\ n) = 2 * \mathcal{O}(log_2\ n) = \mathcal{O}(log_2\ n)$
-El tercer caso es cuando se quiera definir una clave que no estaba definida anteriormente. Nuevamente ignoraremos las asignaciones y condicionales, y nos centraremos en la creación del arreglo "camino", y los siguientes dos ciclos. Dado que se quiere crear un arreglo donde se guarden los punteros a nodos recorridos, como máximo en éste se guardaran $log_2\ n + 1$ nodos (porque en peor caso tendríamos que descender hasta la hoja más lejana para insertar). Por eso, basta con crear el arreglo con $log_2\ n + 1$ posiciones, y esto cuesta $\mathcal{O}(log_2\ n)$. Luego, el primer ciclo consiste en iterar hasta llegar a la posición donde queremos insertar el nuevo nodo, nuevamente esto es $\mathcal{O}(log_2\ n)$ porque en peor caso tendríamos que descender hasta la hoja más lejana. Por último, el último ciclo recorre el arreglo "camino" de atrás hacia adelante (en realidad no todo el arreglo, sino desde el último elemento insertado en él), y dado que éste tiene (a lo sumo) $log_2\ n$ elementos, esto es $\mathcal{O}(log_2\ n)$. Finalmente, dado que este caso tiene éstas tres complejidades no anidadas: $\mathcal{O}(log_2\ n) + \mathcal{O}(log_2\ n) + \mathcal{O}(log_2\ n) = 3 * \mathcal{O}(log_2\ n) = \mathcal{O}(log_2\ n)$.
-Ahora, como teníamos tres casos, la complejidad es el máximo de ellos: max($\mathcal{O}(1)$, $\mathcal{O}(log_2\ n)$, $\mathcal{O}(log_2\ n)$) = $\mathcal{O}(log_2\ n)$

```
IBorrar(in c: α, in/out d: diccRapido(α,β))
1:  var pNodo: puntero(nodo) ← d.raiz                                                    O(1)
2:  var camino: arreglo[⌊log2 (d.tam)⌋ + 1] de puntero(nodo)                             O(log2 n)
3:  var nroCamino: nat ← 0                                                               O(1)
4:  camino[nroCamino] ← pNodo                                                            O(1)
5:  while c != *(pNodo).clave do                                                         O(log2 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:     end if
38: else
39:     if *(pNodo).izq == NULL ∧ *(pNodo).der != NULL then                              O(1)
40:         if *(pNodo).padre == NULL then                                               O(1)
41:             *(*(pNodo).der).padre ← NULL                                             O(1)
42:             d.raiz ← *(pNodo).der                                                    O(1)
43:             delete pNodo                                                             O(1)
44:         else
45:             if *(pNodo).clave == *(*(*(pNodo).padre).izq).clave then                 O(1)
46:                 *(*(pNodo).padre).izq ← *(pNodo).der                                 O(1)
47:             else
48:                 *(*(pNodo).padre).der ← *(pNodo).der                                 O(1)
49:             end if
50:             *(*(pNodo).der).padre ← *(pNodo).padre                                   O(1)
51:             delete pNodo                                                             O(1)
52:         end if
53:     else
54:         if *(pNodo).izq != NULL ∧ *(pNodo).der == NULL then
55:             if *(pNodo).padre == NULL then                                           O(1)
56:                 *(*(pNodo).izq).padre ← NULL                                         O(1)
```

```
57:                 d.raiz ← *(pNodo).izq                                                                    O(1)
58:                 delete pNodo                                                                             O(1)
59:             else
60:                 if *(pNodo).clave == *(*(*(pNodo).padre).izq).clave then                                 O(1)
61:                     *(*(pNodo).padre).izq ← *(pNodo).izq                                                 O(1)
62:                 else
63:                     *(*(pNodo).padre).der ← *(pNodo).izq                                                 O(1)
64:                 end if
65:                 *(*(pNodo).izq).padre ← *(pNodo).padre                                                   O(1)
66:                 delete pNodo                                                                             O(1)
67:             end if
68:         else
69:             if *(pNodo).padre == NULL then                                                               O(1)
70:                 var nuevoPNodo: puntero(nodo) ← *(pNodo).der                                             O(1)
71:                 nroCamino ← nroCamino + 1                                                                O(1)
72:                 camino[nroCamino] ← nuevoPNodo                                                           O(1)
73:                 while *(nuevoPNodo).izq != NULL do                                                       O(log_2 n)
74:                     if *(nuevoPNodo).der == NULL then                                                    O(1)
75:                         *(nuevoPNodo).alt ← *(nuevoPNodo).alt - 1                                        O(1)
76:                     else
77:                         *(nuevoPNodo).alt ← max(*(*(nuevoPNodo).izq).alt - 1, *(*(nuevoPNodo).der).alt)  O(1)
78:                     end if
79:                     nuevoPNodo ← *(nuevoPNodo).izq                                                       O(1)
80:                     nroCamino ← nroCamino + 1                                                            O(1)
81:                     camino[nroCamino] ← nuevoPNodo                                                       O(1)
82:                 end while
83:                 *(pNodo).clave ← *(nuevoPNodo).clave                                                     O(1)
84:                 *(pNodo).significado ← *(nuevoPNodo).significado                                         O(1)
85:                 if *(nuevoPNodo).der != NULL then                                                        O(1)
86:                     if *(*(*(nuevoPNodo).padre).izq).clave == *(nuevoPNodo).clave then                   O(1)
87:                         *(*(nuevoPNodo).padre).izq ← *(nuevoPNodo).der                                   O(1)
88:                     else
89:                         *(*(nuevoPNodo).padre).der ← *(nuevoPNodo).der                                   O(1)
90:                     end if
91:                     *(*(nuevoPNodo).der).padre ← *(nuevoPNodo).padre                                     O(1)
92:                 else
93:                     if *(*(*(nuevoPNodo).padre).izq).clave == *(nuevoPNodo).clave then                   O(1)
94:                         *(*(nuevoPNodo).padre).izq ← NULL                                                O(1)
95:                     else
96:                         *(*(nuevoPNodo).padre).der ← NULL                                                O(1)
97:                     end if
98:                 end if
99:                 delete nuevoPNodo                                                                        O(1)
100:            else
101:            end if
102:        end if
103:    end if
104: end if
105: d.tam ← d.tam - 1                                                                                       O(1)
106: nroCamino ← nroCamino - 1                                                                               O(1)
107: while nroCamino ≥ 0 do                                                                                  O(log_2 n)
108:     pNodo ← camino[nroCamino]                                                                           O(1)
109:     if |FactorDesbalance(pNodo)| > 1 then                                                               O(1)
110:         Rotar(pNodo)                                                                                    O(1)
111:     else
112:     end if
113:     nroCamino ← nroCamino - 1                                                                           O(1)
```

114: **end while**

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

Siendo n la cantidad de nodos.
Nuevamente ignoraremos los condicionales y asignaciones (dado que son constantes), y nos centraremos en la creación de "camino" , y los posteriores tres ciclos.
-La creación de "camino", como hemos visto toma $\mathcal{O}(log_2\ n)$, dado que es crear un array con (a lo sumo) $log_2\ n + 1$ posiciones (esto es el camino recorrido, a rebalancear).
-El primer ciclo consiste en buscar el elemento a borrar, dado que es una búsqueda en un AVL, esto es $\mathcal{O}(log_2\ n)$.
-El segundo ciclo sucede sólo cuando el elemento a borrar tiene dos subárboles hijos distintos de NULL, esto consiste en buscar el sucesor in-order (es decir, bajar un nodo a la derecha, y luego bajar lo máximo posible hacia la izquierda. Así se encuentra el siguiente "inmediato"). Dado que es una búsqueda, y se empieza a descender desde el nodo a borrar (en peor caso, se empieza desde la raíz), esto toma $\mathcal{O}(log_2\ n)$. Cabe aclarar que éste ciclo no siempre se ejecuta, pero dado que en los demás casos la complejidad es de $\mathcal{O}(1)$, podemos asumir que dado el caso que haya sido, estará acotado por la complejidad del peor, osea éste.
-El tercer ciclo consiste en recorrer los elementos de "camino" de atrás hacia adelante (e ir rotando según corresponda), esto toma $\mathcal{O}(log_2\ n)$.
-Finalmente, la complejidad total es la suma de todos estas complejidades parciales: $\mathcal{O}(log_2\ n) + \mathcal{O}(log_2\ n) + \mathcal{O}(log_2\ n) + \mathcal{O}(log_2\ n) = 4 * \mathcal{O}(log_2\ n) = \mathcal{O}(log_2\ n)$

IVacío?(**in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : `bool`

1: **if** d.raiz == NULL **then** $\mathcal{O}(1)$
2: $res \leftarrow$ true $\mathcal{O}(1)$
3: **else**
4: $res \leftarrow$ false $\mathcal{O}(1)$
5: **end if**

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

IClaveMax(**in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : α

1: var pNodo: puntero(nodo) $\leftarrow$ d.raiz $\mathcal{O}(1)$
2: **while** *(pNodo).der != NULL **do** $\mathcal{O}(log_2\ n)$
3: pNodo $\leftarrow$ *(pNodo).der $\mathcal{O}(1)$
4: **end while**
5: $res \leftarrow$ *(pNodo).clave $\mathcal{O}(1)$

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

Siendo n la cantidad de nodos. Ignorando las asignaciones, vemos que lo único a calcular es la cantidad de iteraciones del ciclo. Dado que el ciclo es una búsqueda en un AVL (en particular, se busca el elemento más grande), éste tomará a lo sumo $log_2\ n$ iteraciones.

IClaves(**in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : `itClave`

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

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

ICrearIt(**in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : `itClave`

1: $res \leftarrow$ tupla(1, 0, d.tam, d.raiz, d.raiz) $\mathcal{O}(1)$

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

IHayMas?(**in** *it*: `itClave`) → *res* : `bool`

```
1: if it.1, < it.2 - 1 then                                                    O(1)
2:     res ← true                                                              O(1)
3: else
4:     res ← false                                                             O(1)
5: end if
```

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

IActual(**in** *it*: `itClave`) → *res* : α

```
1: res ← *(it.3).clave                                                         O(1)
```

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

IAvanzar(**in/out** *it*: `itClave`)

```
1: it.1 ← it.1 + 1                                                             O(1)
2: var itNodosNivelActual ← crearIt(DameNodos(it.4, 1, it.0))                  O(n)
3: var bAvanzar:bool ← true                                                    O(1)
4: while bAvanzar do                                                           O(n)
5:     Avanzar(itNodosNivelActual)                                             O(1)
6:     if Anterior(itNodosNivelActual) == Actual(it) then                      O(1)
7:         bAvanzar ← false                                                    O(1)
8:     else
9:     end if
10: end while
11: if HaySiguiente?(itNodosNivelActual) then                                  O(1)
12:     it.3 ← Siguiente(itNodosNivelActual)                                   O(1)
13: else
14:     it.0 ← it.0 + 1                                                        O(1)
15:     it.3 ← Siguiente(crearIt(DameNodos(it.4, 1, it.0)))                    O(n)
16: end if
```

Complejidad: $\mathcal{O}(n2)$

Siendo n la cantidad de nodos.
el ciclo busca encontrar el nodo en donde se encuentra el iterador, para eso avanza el nuevo iterador creado, que itera un conjunto de nodos -estos nodos son todos los del nivel al que pertence el iterador buscado-. En el peor caso este conjunto es de n / 2 elementos, porque sería el nivel más bajo. Por eso tiene complejidad $\mathcal{O}(n)$.
Además se le agrega a la complejidad total, la complejidad de llamar a DameNodos dos veces.
La complejidad total sería: $\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) = 3 * \mathcal{O}(n) = \mathcal{O}(n)$

IDameNodos(**in** *p*: `puntero(nodo)`, **in** *actual*: `nat`, **in** *destino*: `nat`) → *res* : `Conj(nodo)`

```
1: res ← Vacío()                                                               O(1)
2: if p == NULL then                                                           O(1)
3: else
4:     if actual == destino then                                               O(1)
5:         AgregarAtrás(res , p)                                               O(1)
6:     else
7:         Union(DameNodos(*(p).izq, actual + 1, destino), DameNodos(*(p).der, actual + 1, destino))  O(n)
8:     end if
9: end if
```

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

```
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)`)

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