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 $\{n < long(s)\}$

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$\{e_1, e_2 \subset$ estaciones(m) $\wedge_L$ 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 $\{\neg$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`, *destinos*: `diccString` (*compu*: `string`, *caminosMinimos*: `conj(lista(compu))`))
, *computadoras*: `conj(compu)`)

2.4. InvRep y Abs

1. claves(*directasEInterfaces*) es igual a claves(*deOrigenADestino*), al conjunto formado por las ip de *computadoras* y también a claves(*directas*).
2. Si def?(c, *directas*) de alguna clave de *directasEInterfaces* entonces c pertenece al conjunto formado por las ip de *computadoras*.
3. Para todo c, si def?(c, *directasEInterfaces*), entonces ¬def?(c, obtener(c, *directasEInterfaces*).*directas*).
4. Para todo c_1 def?(c_1, *directasEInterfaces*) $\wedge_{\text{L}}$ def?(c_2, Obtener(c_1, *directasEInterfaces*).*directas*) $\Leftrightarrow$ def?(c_2, *directasEInterfaces*) $\wedge_{\text{L}}$ def?(c_1, Obtener(c_2, *directasEInterfaces*).*directas*).
5. Los significados de *directas* son únicos.
6. Si def?(c, *destinos*) de alguna clave de *deOrigenADestino* entonces c pertenece al conjunto formado por las ip de *computadoras*.
7. Para todo c, si def?(c, *deOrigenADestino*), entonces ¬def?(c, obtener(c, *deOrigenADestino*)).
8. Para todo c, si def?(c, *deOrigenADestino*), entonces Claves(Obtener(c, *deOrigenADestino*)) es igual al conjunto formado por las ip de *computadoras* menos la ip de c.
9. Si c pertenece a alguna lista de *significados* de *destinos* para cualquier clave, entonces c pertenece a *computadoras*.

Rep : red $\longrightarrow$ bool
Rep(r) $\equiv$ true $\Longleftrightarrow$

1. claves(e.$directasEInterfaces$) = claves(e.$deOrigenADestino$) = conjips(e.$computadoras$) $\wedge$ (($\forall$ c:ip) (def?(c, e.$directasEInterfaces$) $\Rightarrow_{\text{L}}$ Claves(Obtener(c, e.$directasEInterfaces$).$directas$) = conjips(e.$computadoras$))) $\wedge$ (($\forall$ c:ip) (def?(c, e.$deOrigenADestino$) $\Rightarrow_{\text{L}}$ Claves(Obtener(c, e.$deOrigenADestino$)) = conjips(e.$computadoras$))) $\wedge_{\text{L}}$
2. ($\forall$ c_1, c_2:ip) (def?(c_1, e.$directasEInterfaces$) $\wedge_{\text{L}}$ def?(c_2, Obtener(c_1, e.$directasEInterfaces$).$directas$) $\Rightarrow_{\text{L}}$ $c_2 \in$ conjips(e.$computadoras$)) $\wedge_{\text{L}}$
3. ($\forall$ c:ip) (def?(c, e.$directasEInterfaces$) $\Rightarrow_{\text{L}}$ $\neg$def?(c, Obtener(c, e.$directasEInterfaces$).$directas$)) $\wedge_{\text{L}}$
4. ($\forall$ c_1, c_2:ip) (def?(c_1, e.$directasEInterfaces$) $\wedge_{\text{L}}$ def?(c_2, Obtener(c_1, e.$directasEInterfaces$).$directas$) $\Leftrightarrow$ def?(c_2, e.$directasEInterfaces$) $\wedge_{\text{L}}$ def?(c_1, Obtener(c_2, e.$directasEInterfaces$).$directas$)) $\wedge_{\text{L}}$
5. ($\forall$ c_1, c_2, c_3:ip) ($c_2 \neq c_3 \wedge$ def?(c_1, e.$directasEInterfaces$) $\wedge_{\text{L}}$ def?(c_2, Obtener(c_1, e.$directasEInterfaces$).$directas$) $\wedge$ def?(c_3, Obtener(c_1, e.$directasEInterfaces$).$directas$) $\Rightarrow_{\text{L}}$ obtener(c_2, Obtener(c_1, e.$directasEInterfaces$).$directas$) $\neq$ obtener(c_3, Obtener(c_1, e.$directasEInterfaces$).$directas$)) $\wedge_{\text{L}}$
6. ($\forall$ c:ip) (def?(c, e.$deOrigenADestino$) $\Rightarrow_{\text{L}}$ Claves(Obtener(c, e.$deOrigenADestino$)) $\subseteq$ conjips(e.$computadoras$)) $\wedge_{\text{L}}$
7. ($\forall$ c:ip)(def?(c, e.$deOrigenADestino$) $\rightarrow$ $\neg$def?(c, Obtener(c, e.$deOrigenADestino$)) $\wedge_{\text{L}}$
8. ($\forall$ c_1, c_2, c_3:compu) (def?($c_1.ip$, e.$deOrigenADestino$) $\wedge_{\text{L}}$ def?($c_2.ip$, Obtener(c_1, e.$deOrigenADestino$)) $\wedge_{\text{L}}$ Pertenece?(c_3, Obtener($c_2.ip$, Obtener(c_1, e.$deOrigenADestino$)) $\Rightarrow_{\text{L}}$ $c_3 \in$ e.$computadoras$)

Abs : e_red e $\longrightarrow$ red {Rep(e)}
Abs(e) $=_{\text{obs}}$ r: red |
computadoras(r) = e.computadoras $\wedge_{\text{L}}$
($\forall$ c_1, c_2:compu) ($c_1 \in$ computadoras(r) $\wedge$ $c_2 \in$ computadoras(r) $\Rightarrow_{\text{L}}$ (Def?(c_1, e.$directasEInterfaces$) $\wedge_{\text{L}}$ Def?(c_2, Obtener(c_1, e.$directasEInterfaces$).$directas$ = conectadas?(r, c_1, c_2))) $\wedge_{\text{L}}$
($\forall$ c_1, c_2:compu) (conectadas?(r, c_1, c_2) $\Rightarrow_{\text{L}}$ (Obtener(c_2, Obtener(c_1, e.$directasEInterfaces$).$directas$) = interfazUsada(r, c_1, c_2)))

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: $res \leftarrow$ Def?(Obtener($r.directasEInterfaces$, $c_1.ip$).$directas$, $c_2.ip$) $\mathcal{O}(|c_2.ip| + |c_1.ip|)$

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

iInterfazUsada(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) → res : `interfaz`

1: $res \leftarrow$ Obtener(Obtener($r.directasEInterfaces$, $c_1.ip$).$directas$, $c_2.ip$) $\mathcal{O}(|c_1.ip| + |c_2.ip|)$

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

iIniciarRed() → 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: Definir($r.directasEInterfaces$, $c.ip$, tupla(Vacío(), Vacio())) $\mathcal{O}(|c.ip|)$
2: var $itComputadoras$:itConj(α) ← CrearIt($r.computadoras$) $\mathcal{O}(1)$
3: **while** HaySiguiente?($itComputadoras$) **do** $\mathcal{O}(1)$
4: Definir(Obtener($r.deOrigenADestino$, Siguiente($itComputadoras$).ip), $c.ip$, Vacío()) $\mathcal{O}(|Siguiente(\text{itComputadoras}).i$
$|c.ip|)$
5: Avanzar($itComputadoras$) $\mathcal{O}(1)$
6: **end while**
7: var $dicNuevaCompu$:dicString(α) ← Vacio() $\mathcal{O}(1)$
8: **while** HayAnterior?($itComputadoras$) **do** $\mathcal{O}(1)$
9: Definir($dicNuevaCompu$, Anterior($itComputadoras$).ip, Vacío()) $\mathcal{O}(1)$
10: Retroceder($itComputadoras$) $\mathcal{O}(1)$
11: **end while**
12: Definir($r.deOrigenADestino$, $c.ip$, $dicNuevaCompu$) $\mathcal{O}(1)$
13: Agregar($r.computadoras$, c) $\mathcal{O}(1)$

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 ← Obtener($r.directasEInterfaces$, $c_1.ip$)
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 ← Obtener($r.directasEInterfaces$, $c_2.ip$)
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: var $itOrigenes$:itConj(α) ← CrearIt($r.computadoras$) $\mathcal{O}(1)$
8: **while** HaySiguiente?($itOrigenes$) **do** $\mathcal{O}(1)$
9: var $dicCompu$:dicString(α) ← Obtener($r.deOrigenADestino$, Siguiente($itOrigenes$).ip) $\mathcal{O}(1)$
10: var $itDestinos$:itConj(α) ← CrearIt($r.computadoras$) $\mathcal{O}(1)$
11: **while** HaySiguiente?($itDestinos$) **do** $\mathcal{O}(1)$
12: **if** Siguiente($itOrigenes$) <> Siguiente($itDestinos$)) **then** $\mathcal{O}(1)$
13: Definir($dicCompu$, Siguiente($itDestinos$).ip), CalcularCaminosMinimos(r, Siguiente($itOrigenes$),
Siguiente($itDestinos$))) $\mathcal{O}(1)$
14: **end if**
15: Avanzar($itDestinos$) $\mathcal{O}(1)$
16: **end while**
17: Avanzar($itOrigenes$) $\mathcal{O}(1)$
18: **end while**

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: $res \leftarrow$ Obtener($r.directasEInterfaces$, $c.ip$).$compusDirectas$

Complejidad:

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

```
1: var tupVecinos:tupla ← Obtener(r.directasEInterfaces, c.ip)                          O(1)
2: var itcompusDirectas:itConj(compu) ← CrearIt(tupVecinos.compusDirectas)              O(1)
3: res:bool ← false                                                                     O(1)
4: while HaySiguiente(itcompusDirectas) AND ¬res do                                     O(1)
5:     if Obtener(tupVecinos.directas, Siguiente(itcompusDirectas).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: $res \leftarrow$ CrearIt(Obtener(Obtener($r.deOrigenADestino$, $c_1.ip$), $c_2.ip$)) $\mathcal{O}(1)$

Complejidad:

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

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

Complejidad:

iCalcularCaminosMinimos(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `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** 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.

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

El dcnet esta representado con una tupla en la cual: vamos a tener la red (en d.red) con la que se inicia el dcnet(pasada por referencia), una tupla que tiene la computadora que mas paquetes envio y cuantos ha enviado (en d.MasEnviante). Luego, un diccionario String (d.CompYPaq) cuyas claves son las computadoras de la red y su significado es una tupla. Esta tupla tiene tres componentes: un Diccionario Rapido (MasPriori) cuya clave es un natural (prioridad, osea la prioridad del paquete) y su significado es un conj(paquete) (PaqdePriori), luego otro Diccionario Rapido (PaqYCam) cuya clave es el paquete (habiendo establecido previamente que la relacion de orden entre paquetes es por id) y su significado es una secu(compu) (CamRecorrido), finalmente un natural (Enviados) que representa la cantidad de paquetes enviados por la computadora.

`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*)), *Enviados*:nat)
)

3.3. InvRep y Abs

1. El conjunto armado por las ips de las computadoras de 'red' es igual al conjunto con todas las claves de 'CompYPaq'.
2. Para toda compu (c1) perteneciente a las claves de 'CompYPaq' cuyo significado.enviados es (e1), d.MasEnviante.enviados (e2) es mayor o igual a e1. Luego, d.MasEnviante.compu esta definida en 'CompYPaq' y su significado es e2.
3. si una computadora pertenece a un significado de 'PaqYCam', entonces pertenece a claves de 'CompYPaq'
4. La union de todos los significados de MasPriori es igual a las claves de 'PaqYCam'
5. si un paquete (paq) pertenece al significado de una prioridad (pri) en el diccionario MasPriori, entonces la prioridad de paq es pri
6. no existen claves en el diccionario 'MasPriori' con significado vacio.
7. si un paquete (p1) esta definido en 'PaqYCam' como significado de una computadora (c1) en el diccionario 'CompYPaq', entonces p1 no puede estar definido en 'PaqYCam' como significado de una computadora (c2), siendo c1 y c2 distintas.
8. si un paquete (p1) esta definido en 'PaqYCam' como significado de una computadora(c1) en el diccionario 'CompYPaq', entonces la ultima componente de su significado (osea, 'CamRecorrido') es c1

Rep : e_dc $\longrightarrow$ bool

Rep(d) $\equiv$ true $\iff$ claves(d.CompYPaq) = ipCompus(computadoras(d.red))$\wedge$ 1
($\forall$ c:compu)(c.ip $\in$ claves(d.CompYPaq))
(obtener(c.ip,d.CompYPaq).enviados $\leq$ (d.MasEnviante).enviados $\wedge$
def?(d.MasEnviante).compu, d.CompYPaq) $\wedge_{\text{L}}$
obtener(d.MasEnviante.compu, d.CompYPaq).enviados = (d.MasEnviante).enviados) 2
($\forall$ c_2:compu, p:paquete)(esta?(c_2,obtener(p,obtener(c.ip, d.CompYPaq).PaqYCam)) $\Rightarrow$
def?(c_2, d.CompYPaq) 3
$\wedge$ juntarSignificados(obtener(c.ip,d.CompYPaq).MasPriori,claves(obtener(c.ip,d.CompYPaq).MasPri
= claves(obtener(c.ip,d.CompYPaq).PaqYCam) 4
($\forall$ pr:Nat,p:paquete)(def?(pr,Obtener(c.ip,d.CompYPaq).MasPriori) $\wedge_{\text{L}}$
p $\in$ obtener(pr,c.ip,d.CompYPaq).MasPriori)$\rightarrow$ pr = p.prioridad) $\wedge$ 5
($\forall$ pr:Nat) (Def?(pr, Obtener(c.ip, d.CompYPaq).MasPriori) $\Rightarrow_{\text{L}}$ $\neg$ $\emptyset$? Obtener(pr, Obtener(c.ip, d.CompYPaq).MasPriori)) $\wedge$ 6
($\forall$ p1:paquete, c2:computadora) (Def?(p, Obtener(c.ip, d.CompYPaq).PaqYCam) $\wedge$
def?(c2.ip, d.CompYPaq) $\Rightarrow_{\text{L}}$ $\neg$def?(p, Obtener(c2.ip, d.CompYPaq).PaqYCam) $\wedge$ 7
($\forall$ p:paquete) (def?(p, obtener(c.ip, d.CompYPaq).PaqYCam) $\Rightarrow_{\text{L}}$ ult(obtener(p, obtener(c.ip, d.CompYPaq).PaqYCam)) = c) 8

ipCompus : Conj(compu) $\longrightarrow$ Conj(string)
juntarSignificados Dicc $\times$ Conj $\longrightarrow$ Conj

juntarSignificados(dic, cl) $\equiv$
if vacia?(cl) **then**
$\emptyset$
else
obtener(DameUno(cl),dic) $\cup$ juntarSignificados(dic, SinUno(cl))
fi
ipCompus(cc) $\equiv$ **if** vacia?(cp) **then** $\emptyset$ **else** Agregar(DameUno(cc).ip,ipCompus(SinUno(cc))) **fi**

Abs : e_dc e $\longrightarrow$ dcnet {Rep(e)}
Abs(e) $=_{\text{obs}}$ d: dcnet |
red(d) = e.red $\wedge_{\text{L}}$
($\forall$ p:paquete, c:compu) (paqueteEnTransito?(d, p) $\Rightarrow_{\text{L}}$ (def?(c.ip, e.CompYPaq $\wedge_{\text{L}}$ def?(p, obtener(c.ip, e.CompYPaq).PaqYCam) $\Rightarrow_{\text{L}}$ obtener(p, obtener(c.ip, e.CompYPaq).PaqYCam) = caminoRecorrido(d, p)))
($\forall$ c:compu) (c $\in$ computadoras(red(d)) $\Rightarrow_{\text{L}}$ ((def?(c.ip, e.CompYPaq) $\Rightarrow_{\text{L}}$ obtener(c.ip, e.CompYPaq).Enviados = cantidadEnviados(d, c)) $\wedge$ (def?(c.ip, e.CompYPaq) $\Rightarrow_{\text{L}}$ claves(obtener(c.ip, e.CompYPaq).PaqYCam) = enEspera(d, c))))

3.4. Algoritmos

Algoritmos

iRed(**in** d: `dcnet`) $\rightarrow$ res : `red`
1: $res \leftarrow (d.\text{red})$ $\mathcal{O}(1)$

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

iCaminoRecorrido(**in** d: `dcnet`,**in** p: `paquete`) $\rightarrow$ res : `secu(compu)`
1: var $it \leftarrow$ creaIt(d.CompYPaq) $\mathcal{O}(N)$

2: var *esta*: bool ← false — $\mathcal{O}(1)$
3: **while** HaySiguiente(*it*) ∧ ¬ esta **do** — $\mathcal{O}(n)$
4: var *diccpaq*:diccRapido ← ((Siguiente(*it*)).significado).PaqYCam — $\mathcal{O}(1)$
5: **if** def?(*p*,*diccpaq*) **then** — $\mathcal{O}(log_2(k))$
6: *esta* ← true — $\mathcal{O}(1)$
7: *res* ← obtener(*p*, *diccpaq*) — $\mathcal{O}(log_2(k))$
8: **end if**
9: Avanzar(*it*) — $\mathcal{O}(1)$
10: **end while**

Complejidad: $\mathcal{O}(N * log_2(k))$ Donde N es la cantidad de computadoras en la red, y k la cantidad maxima de paquetes que hay en una computadora.

$\mathcal{O}(1) + \mathcal{O}(1) + n * (\mathcal{O}(1) + \mathcal{O}(log_2(k)) + \mathcal{O}(1) + \mathcal{O}(log_2(k))) =$
$\mathcal{O}(2) + n * (2\mathcal{O}(log_2(k))) =$
$\mathcal{O}(n * (log_2(k)))$

iCantidadEnviados(**in** *d*: `dcnet`, **in** *c*: `compu`) → *res* : `nat`
1: *res* ← 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* ← 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* ← 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.

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

iCrearPaquete(**in** *p* : `paquete`, **in/out** *d* : `dcnet`)
1: var *diccprio*: diccRapido ← obtener(*p*.origen, *d*.CompYPaq).MasPriori) — $\mathcal{O}(L)$
2: var *dicccam*: diccRapido ← obtener(*p*.origen, *d*.CompYPaq).PaqYCam) — $\mathcal{O}(L)$
3: **if** ¬ def?(p.prioridad, *diccprio*) **then** — $\mathcal{O}(log_2(k))$
4: Definir(*p*.prioridad,Agregar(Vacio(), *p*), *diccprio*) — $\mathcal{O}(log_2(k))$
5: **else**
6: Definir(*p*.prioridad,Agregar(obtener(*p*.prioridad, *diccprio*), *p*), *diccprio* — $\mathcal{O}(log_2(k))$
7: **end if**
8: definir(*p*, *dicccam*, AgregarAtras(<>,p.origen) — $\mathcal{O}(log_2(k))$

Complejidad: $\mathcal{O}(L + log_2(k))$ Donde L es la longitud de la computadora de origen del paquete, y k la cantidad de paquetes que EnEspera en esa computadora.

$\mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(log_2(k)) + \mathcal{O}(log_2(k)) + \mathcal{O}(log_2(k)) + \mathcal{O}(log_2(k)) =$
$2 * \mathcal{O}(L) + 4 * \mathcal{O}(log_2(k)) =$
$\mathcal{O}(L + log_2(k))$

IAvanzarSegundo(**in/out** d : dcnet)

1: var $it \leftarrow$ computadoras(red) $\mathcal{O}(1)$
2: var aux $\leftarrow$ Vacia() $\mathcal{O}(1)$
3: **while** HaySiguiente(it) **do** $\mathcal{O}(N)$
4: var *diccprio*: diccRapido $\leftarrow$ obtener(Siguiente(it).id, d.CompYPaq).MasPriori) $\mathcal{O}(L)$
5: var *dicccam*: diccRapido $\leftarrow$ obtener(Siguiente(it).id, d.CompYPaq).PaqYCam) $\mathcal{O}(L)$
6: **if** $\neg$ Vacio?(*diccprio*) **then** $\mathcal{O}(1)$
7: var *paq*: paquete $\leftarrow$ Primero(obtener(ClaveMax(*diccprio*), *diccprio*)) $\mathcal{O}(log_2(k) + 1 + 1)$
8: AgregarAdelante(aux, tupla(paq: *paq*,pcant: it.id, camrecorrido: obtener(*paq*, *dicccam*)) $\mathcal{O}(1 + log_2(k))$
9: Eliminar(obtener(ClaveMax(*diccprio*), *diccprio*), *paq*) $\mathcal{O}(log_2(k) + log_2(k) + 1)$
10: **if** EsVacio?(obtener(ClaveMax(*diccprio*), *diccprio*) **then** $\mathcal{O}(log_2(k))$
11: borrar(ClaveMax(*diccprio*), *diccprio*) $\mathcal{O}(log_2(k))$
12: **end if**
13: borrar(*paq*, *dicccam*) $\mathcal{O}(log_2(k))$
14: obtener(Siguiente(it).id, d.CompYPaq).Enviados ++ $\mathcal{O}(L)$
15: **if** obtener(Siguiente(it).id, d.CompYPaq).Enviados >(d.MasEnviante).enviados **then** $\mathcal{O}(L + 1)$
16: d.MasEnviante $\leftarrow$ tupla(Siguiente(it), obtener(Siguiente(it).id, d.CompYPaq).Enviados) $\mathcal{O}(L + 1)$
17: **end if**
18: **end if**
19: Avanzar(it) $\mathcal{O}(1)$
20: **end while**
21: var *itaux* $\leftarrow$ crearIt(aux) $\mathcal{O}(1)$
22: **while** HaySiguiente(*itaux*) **do** $\mathcal{O}(Nk)$
23: var *proxpc*: compu $\leftarrow$ Primero(Siguiente(CaminosMinimos(d.red, *itaux*.pcant, *itaux*.destino)) $\mathcal{O}(L_1 + L_2)$
24: var *diccprio*: diccRapido $\leftarrow$ obtener(*proxpc*.id, d.CompYPaq).MasPriori) $\mathcal{O}(L)$
25: var *dicccam*: diccRapido $\leftarrow$ obtener(*proxpc*.id, d.CompYPaq).PaqYCam) $\mathcal{O}(L)$
26: **if** *proxpc* $\neq$ (*itaux*.paq).destino **then**
27: **if** def?((*itaux*.paq).prioridad, *diccprio*) **then** $\mathcal{O}(log_2(k))$
28: var *mismaprio*: conj(paquetes) $\leftarrow$ Agregar(obtener(it3.paq.prioridad, *diccprio*), it3.paq) $\mathcal{O}(log_2(k))$
29: definir((it3.paq).prioridad, *mismaprio*, *diccprio*) $\mathcal{O}(log_2(k))$
30: **else**
31: Definir(it3.prioridad, Agregar(Vacio(),it3.paq), *diccprio*) $\mathcal{O}(log_2(k))$
32: **end if**
33: definir(p.paq,AgregarAtras(it3.camrecorrido ,*proxpc*), *dicccam*) $\mathcal{O}(log_2(k))$
34: **end if**
35: EliminarSiguiente(it3) $\mathcal{O}(1)$
36: Avanzar(it3) $\mathcal{O}(1)$
37: **end while**

Complejidad: $\mathcal{O}(N * (L + log_2(k))$ Donde N es la cantidad de computadoras, L la longitud del nombre mas largo de las computadoras, y k la cantidad mas grande de paquetes que tiene una computadora.

$\mathcal{O}(1) + \mathcal{O}(1) + N * (\mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(1) + \mathcal{O}(log_2(k) + 1 + 1) + \mathcal{O}(1 + log_2(k)) + \mathcal{O}(log_2(k) + log_2(k) + 1) +$
$\mathcal{O}(log_2(k)) + \mathcal{O}(log_2(k)) + \mathcal{O}(log_2(k)) + \mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(1)) + N * (\mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(log_2(k)) +$
$\mathcal{O}(log_2(k)) + \mathcal{O}(log_2(k)) + \mathcal{O}(log_2(k))\mathcal{O}(log_2(k)) + \mathcal{O}(1) + \mathcal{O}(1)) =$
$N * (\mathcal{O}(5 * L + 5 * log_2(k))) + N * \mathcal{O}(3 * L + 5 * log_2(k)) =$
$2N * (\mathcal{O}(L + log_2(k)) =$
$\mathcal{O}(N * (L + log_2(k)))$

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(N)
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: $\mathcal{O}(N * log(k))$ Donde N es la cantidad de computadoras en la red y k la cantidad maxima de paquetes que hay en alguna compu.

$\mathcal{O}(1) + \mathcal{O}(1) + N * (\mathcal{O}(log_2(k)) + \mathcal{O}(1)) + \mathcal{O}(1) =$
$\mathcal{O}(3) + N * (\mathcal{O}(log_2(k)) =$
$\mathcal{O}(N * log_2(k))$

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

parámetros formales: string, β;

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO).

géneros: `diccString(string, β)`.

Operaciones básicas de Diccionario String(string, β)

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

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

OBTENER(**in** c: `string`, **in** d: `diccString(string, β)`) $\rightarrow$ res : β
Pre $\equiv$ {def?(c, d)}
Post $\equiv$ {$res =_{\text{obs}}$ obtener(c, d)}
Complejidad: $\mathcal{O}(|c|)$
Descripción: Devuelve el significado correspondiente a la clave
Aliasing: Res es modificable si y sólo si d es modificable

DEFINIR(**in** c: `string`, **in** s: β, **in/out** d: `diccString(string, β)`)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(c, s, d_0)}
Complejidad: $\mathcal{O}(|c|)$
Descripción: Define la clave, asociando su significado, al diccionario
Aliasing: Los elementos c y s se pasan por referencia

BORRAR(**in** c: `string`, **in/out** d: `diccString(string, β)`)
Pre $\equiv$ {$d =_{\text{obs}} d_0 \wedge$ def?(c, d_0)}
Post $\equiv$ {$res =_{\text{obs}}$ borrar(c, d_0)}
Complejidad: $\mathcal{O}(|c|)$
Descripción: Borra la clave (y su significado) del diccionario

La representación y los algoritmos del módulo no hacen falta explicitarlos porque están dados por la cátedra. De todas formas, cabe recordar que este módulo se representa sobre un Trie.

5. Diccionario Rápido

5.1. Interfaz

Interfaz

parámetros formales: α, β; α tiene relación de orden

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO), ITERADOR UNIDIRECCIONAL(α)

géneros: `diccRapido(`α`,`β`)`, ITCLAVE(α)

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
Aliasing: Res es modificable si y sólo si d es modificable

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
Aliasing: Los elementos c y s se pasan por referencia.

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 (y su significado) 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
Aliasing: Res es modificable si y sólo si d es modificable

CLAVES(**in** d: `diccRapido(`α`,`β`)`) $\rightarrow$ res : `itClave`
Pre $\equiv$ {true}

Post $\equiv \{res =_{\text{obs}} \text{CrearIt(claves}(d))\}$
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 $\equiv \{res =_{\text{obs}} \text{crearItUni(secuClaves(d))}\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Crea el iterador de claves
Aliasing: Res es devuelto por referencia

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

Actual(**in** it: `itClave`) $\rightarrow$ res : α
Pre $\equiv \{\text{HayMas?}(it)\}$
Post $\equiv \{res =_{\text{obs}} \text{Actual}(it)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el actual del iterador
Aliasing: Res es modificable si y sólo si el Diccionario es modificable

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

5.2. Auxiliares

Operaciones auxiliares de Diccionario

DameNodos(**in** p: `puntero(nodo)`, **in** $actual$: `nat`, **in** $destino$: `nat`) $\rightarrow$ res : `conjLineal(nodo)`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{nodosNivel}(p, actual, destino)\}$
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 $\equiv \{p =_{\text{obs}} p_0 \wedge \text{p != NULL} \wedge_{\text{L}} (|\text{FactorDesbalance(p)}| > 1)\}$
Post $\equiv \{p =_{\text{obs}} \text{rotar}(p_0)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Realiza la rotación pertinente de p, de ser necesario

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

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

Operaciones auxiliares del Iterador

Siguientes(**in** it: `itClave`) $\rightarrow$ res : `Lista Enlazada`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ siguientes(it)}
Complejidad: $\mathcal{O}(it.tam^2)$
Descripción: Devuelve la lista de nodos no iterados hasta el momento. Están ordenados por orden de aparición.
Aliasing: Los nodos de la lista se pasan por referencia

5.3. Representación

Representación

Representación del Diccionario

Como se sabe que la cantidad de claves no está acotada, este diccionario estará representado con un AVL. Cabe destacar, que las claves del diccionario deben contener una relación de orden.

`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`)

Representación del Iterador

El iterador es una tupla con varias componentes, entre ellas, un puntero al nodo actual. Este puntero (it.nodoActual) apunta a NULL cuando ya no quedan nodos por iterar en el último nivel del AVL. El iterador recorre los nodos por nivel.

`itClave` **se representa con** `estr`

donde `estr` es `tupla`(*nivelActual*: `nat`, *#nodosRecorridos*: `nat`, *tam*: `nat`, *nodoActual*: `puntero`(nodo), *raiz*: `puntero`(nodo))
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 y Abs del Diccionario:

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 puede haber un nodo que sea hijo de dos nodos distintos.
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. No hay dos nodos con la misma componente "clave".
8. Para todo nodo, todos los nodos de su subárbol derecho son mayores a él.
9. Para todo nodo, todos los nodos del su subárbol izquierdo son menores que él.
10. 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.
11. Para todo nodo, la diferencia, en módulo, de la altura entre sus subárboles es menor o igual a 1.
12. Siempre existe un camino entre la raíz y cualquier otro nodo.
13. No hay ciclos. Más formalmente, partiendo de un nodo, no se puede volver a pasar por él sin recurrir a la componente padre.

Abs:

$$\begin{array}{lll} \text{Abs} & : \text{estr } e & \longrightarrow \text{Diccionario(Clave, Significado)} \quad \{\text{Rep}(e)\} \\ \text{Abs}(e) =_{\text{obs}} \text{d: Diccionario(Clave, Significado)} \mid & & \\ & & (\forall \text{c: clave}) \\ & & \text{def?(c, d)} = \text{Def?(c, e)} \wedge_{\text{L}} \\ & & \text{obtener(c, d)} = \text{Obtener(c, e)} \end{array}$$

InvRep y Abs del Iterador:

InvRep en lenguaje coloquial:

1. it.tam es igual a la cantidad de nodos del AVL que itera.
2. it.raiz apunta a la raíz del AVL que itera, o es NULL si el AVL es vacío.
3. it.nodoActual apunta a un nodo cualquiera del AVL que itera, o es NULL si el AVL es vacío.
4. it.#nodosRecorridos está entre 0 (inclusive) y it.tam (inclusive).
5. it.nivelActual está entre 1 (inclusive) y $log_2\ it.tam + 1$ (inclusive).
6. El invariante de nodo es el mismo que el invariante de nodo del Diccionario.

Abs:

$$\begin{array}{lll} \text{Abs} & : \text{estr } e & \longrightarrow \text{Iterador Unidireccional}(\alpha) \quad \{\text{Rep}(e)\} \\ \text{Abs}(e) =_{\text{obs}} \text{i: Iterador Unidireccional}(\alpha) \mid & & \\ & & \text{siguientes(i)} = \text{Siguientes(e)} \end{array}$$

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 * k)
3:      if *(pNodo).clave == c then                                    O(k)
4:          res ← true                                                 O(1)
5:          return res                                                 O(1)
6:      else
7:          if c > *(pNodo).clave then                                 O(k)
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: $\mathcal{O}(log_2\ n * k)$

Siendo n la cantidad de nodos y k el costo de comparación de α.
Vamos a ignorar 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 y cuánto cuesta cada una. Como se trata de buscar un nodo en un AVL, sabemos que la búsqueda es $\mathcal{O}(log_2\ n * k)$, 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 $\mathcal{O}(log_2\ n)$ veces. Luego, cada iteración cuesta $\mathcal{O}(k)$. Finalmente, 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 * k)
3:      if c > *(pNodo).clave then                                     O(k)
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: $\mathcal{O}(log_2\ n * k)$

Siendo n la cantidad de nodos y k el costo de comparación de α.
Es un algoritmo muy parecido al de Def?. Nuevamente ignoraremos 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 y cuánto cuesta cada una. Como se trata de buscar un nodo en un AVL, sabemos que la búsqueda es $\mathcal{O}(log_2\ n * k)$, 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 $\mathcal{O}(log_2\ n)$ veces. Luego, cada iteración cuesta $\mathcal{O}(k)$. Finalmente, 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(log_2 n * k)
 7:         var pNodo: puntero(nodo) ← d.raiz                                 O(1)
 8:         while *(pNodo).clave != c do                                      O(log_2 n * k)
 9:             if c > *(pNodo).clave then                                    O(k)
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:         while seguir == true do                                           O(log_2 n * k)
20:             if c > *(pNodo).clave ∧ *(pNodo).der == NULL then             O(k)
21:                 *(pNodo).der ← &tupla(c, s, pNodo, NULL, NULL, 1)         O(1)
22:                 seguir ← false                                            O(1)
23:             else
24:                 if c > *(pNodo).clave ∧ *(pNodo).der != NULL then         O(k)
25:                     pNodo ← *(pNodo).der                                  O(1)
26:                 else
27:                     if c < *(pNodo).clave ∧ *(pNodo).izq == NULL then     O(k)
28:                         *(pNodo).izq ← &tupla(c, s, pNodo, NULL, NULL, 1) O(1)
29:                         seguir ← false                                    O(1)
30:                     else
31:                         pNodo ← *(pNodo).izq                              O(1)
32:                     end if
33:                 end if
34:             end if
35:         end while
36:         d.tam ← d.tam + 1                                                 O(1)
37:         while pNodo != NULL do                                            O(log_2 n * k)
38:             var padrePNodo ← *(pNodo).padre                               O(1)
39:             if |FactorDesbalance(pNodo)| > 1 then                         O(1)
40:                 Rotar(pNodo)                                              O(k)
41:             else
42:                 *(pNodo).alt ← Altura(pNodo)                              O(1)
43:             end if
44:             pNodo ← padrePNodo                                            O(1)
45:         end while
46:     end if
47: end if
```

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

Siendo n la cantidad de nodos, y k el costo de comparación de α.
En este algoritmo, tomaremos en cuenta las complejidades de tres casos e ignoraremos las 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 (cuyas complejidades son $\mathcal{O}(1)$), y nos centraremos en el uso de DEF? y el ciclo. DEF? sabemos que toma $\mathcal{O}(log_2\ n * k)$, y en cuanto al ciclo, sabemos que tomará $\mathcal{O}(log_2\ n * k)$ también, porque iterará hasta buscar el nodo buscado (itera $\mathcal{O}(log_2\ n)$ veces) y cada iteración cuesta $\mathcal{O}(k)$. Esto es: $\mathcal{O}(log_2\ n * k) + \mathcal{O}(log_2\ n * k) = 2 * \mathcal{O}(log_2\ n * k) = \mathcal{O}(log_2\ n * k)$
-El tercer caso es cuando se quiera definir una clave que no estaba definida anteriormente. Nuevamente ignoraremos las asignaciones, y nos centraremos en los dos ciclos. El primer ciclo consiste en iterar hasta llegar a la posición donde queremos insertar el nuevo nodo, nuevamente esto es $\mathcal{O}(log_2\ n * k)$ porque en peor caso tendríamos que descender hasta la hoja más lejana ($\mathcal{O}(log_2\ n)$ iteraciones, y cada iteración cuesta $\mathcal{O}(k)$). Por último, el segundo ciclo recorre, de abajo hacia arriba, la rama por la que acabamos de bajar, y dado que como ya mencionamos que ésta tiene $\mathcal{O}(log_2\ n)$ elementos, esto es $\mathcal{O}(log_2\ n * k)$. Finalmente, dado que este caso tiene éstas tres complejidades no anidadas: $\mathcal{O}(log_2\ n * k) + \mathcal{O}(log_2\ n * k) + \mathcal{O}(log_2\ n * k) = 3 * \mathcal{O}(log_2\ n * k) = \mathcal{O}(log_2\ n * k)$.
-Ahora, como teníamos tres casos, la complejidad es el máximo de ellos: max($\mathcal{O}(1)$, $\mathcal{O}(log_2\ n * k)$, $\mathcal{O}(log_2\ n * k)$) = $\mathcal{O}(log_2\ n * k)$

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

```
1:  var pNodo: puntero(nodo) ← d.raiz                                          O(1)
2:  while c != *(pNodo).clave do                                               O(log2 n * k)
3:      if c > *(pNodo).clave then                                             O(k)
4:          pNodo ← *(pNodo).der                                               O(1)
5:      else
6:          pNodo ← *(pNodo).izq                                               O(1)
7:      end if
8:  end while
9:  if *(pNodo).izq == NULL ∧ *(pNodo).der == NULL then                        O(1)
10:     if *(pNodo).padre == NULL then                                         O(1)
11:         d.raiz ← NULL                                                      O(1)
12:     else
13:         if pNodo == *(*(pNodo).padre).izq then                             O(k)
14:             *(*(pNodo).padre).izq ← NULL                                   O(1)
15:         else
16:             *(*(pNodo).padre).der ← NULL                                   O(1)
17:         end if
18:     end if
19: else
20:     if *(pNodo).izq == NULL ∧ *(pNodo).der != NULL then                    O(1)
21:         if *(pNodo).padre == NULL then                                     O(1)
22:             *(*(pNodo).der).padre ← NULL                                   O(1)
23:             d.raiz ← *(pNodo).der                                          O(1)
24:         else
25:             if pNodo == *(*(pNodo).padre).izq then                         O(k)
26:                 *(*(pNodo).padre).izq ← *(pNodo).der                       O(1)
27:             else
28:                 *(*(pNodo).padre).der ← *(pNodo).der                       O(1)
29:             end if
30:             *(*(pNodo).der).padre ← *(pNodo).padre                         O(1)
31:         end if
32:     else
33:         if *(pNodo).izq != NULL ∧ *(pNodo).der == NULL then                O(1)
34:             if *(pNodo).padre == NULL then                                 O(1)
35:                 *(*(pNodo).izq).padre ← NULL                               O(1)
36:                 d.raiz ← *(pNodo).izq                                      O(1)
37:             else
38:                 if pNodo == *(*(pNodo).padre).izq then                     O(k)
```

```
39:                     *(*(pNodo).padre).izq ← *(pNodo).izq                    O(1)
40:                 else
41:                     *(*(pNodo).padre).der ← *(pNodo).izq                    O(1)
42:                 end if
43:                 *(*(pNodo).izq).padre ← *(pNodo).padre                      O(1)
44:             end if
45:         else
46:             var nuevoPNodo: puntero(nodo) ← *(pNodo).der                    O(1)
47:             while *(nuevoPNodo).izq != NULL do                              O(log_2 n)
48:                 nuevoPNodo ← *(nuevoPNodo).izq                              O(1)
49:             end while
50:             *(pNodo).clave ← *(nuevoPNodo).clave                            O(1)
51:             *(pNodo).significado ← *(nuevoPNodo).significado                O(1)
52:             if *(nuevoPNodo).der != NULL then                               O(1)
53:                 if *(*(nuevoPNodo).padre).izq == nuevoPNodo then            O(k)
54:                     *(*(nuevoPNodo).padre).izq ← *(nuevoPNodo).der          O(1)
55:                 else
56:                     *(*(nuevoPNodo).padre).der ← *(nuevoPNodo).der          O(1)
57:                 end if
58:                 *(*(nuevoPNodo).der).padre ← *(nuevoPNodo).padre            O(1)
59:             else
60:                 if *(*(nuevoPNodo).padre).izq == nuevoPNodo then            O(k)
61:                     *(*(nuevoPNodo).padre).izq ← NULL                       O(1)
62:                 else
63:                     *(*(nuevoPNodo).padre).der ← NULL                       O(1)
64:                 end if
65:             end if
66:         end if
67:     end if
68: end if
69: d.tam ← d.tam - 1                                                           O(1)
70: pNodo ← *(nuevoPNodo).padre                                                 O(1)
71: while pNodo != NULL do                                                      O(log_2 n * k)
72:     var padrePNodo ← *(pNodo).padre                                         O(1)
73:     if |FactorDesbalance(pNodo)| > 1 then                                   O(1)
74:         Rotar(pNodo)                                                        O(k)
75:     else
76:         *(pNodo).alt ← Altura(pNodo)                                        O(1)
77:     end if
78:     pNodo ← padrePNodo                                                      O(1)
79: end while
```

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

Siendo n la cantidad de nodos y k el costo de comparación de α.
Nuevamente ignoraremos los condicionales -fuera de ciclos- (dado que la complejidad de éstos pertenecen a la complejidad de los ciclos) y asignaciones (dado que son constantes), y nos centraremos en los tres ciclos.
-El primer ciclo consiste en buscar el elemento a borrar, dado que es una búsqueda en un AVL, esto es $\mathcal{O}(log_2\ n * k)$.
-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 $\mathcal{O}(log_2\ n)$ (porque sólo se baja hacia la izquierda, y no hay comparaciones de tipo α). Cabe aclarar que éste ciclo no siempre se ejecuta, pero dado que en los demás casos la complejidad es de $\mathcal{O}(k)$, podemos asumir que dado el caso que haya sido, estará acotado por la complejidad del peor, osea éste.
-El tercer ciclo consiste en recorrer, de abajo hacia arriba, la rama en la cual se borró el nodo auxiliar buscado (e ir rotando según corresponda), esto toma $\mathcal{O}(log_2\ n * k)$, porque se itera $\mathcal{O}(log_2\ n)$ veces y cada iteración

cuesta $\mathcal{O}(k)$.
-Finalmente, la complejidad total es la suma de todos estas complejidades parciales: $\mathcal{O}(log_2\ n * k) + \mathcal{O}(log_2\ n) + \mathcal{O}(log_2\ n * k) = \mathcal{O}(log_2\ n * k) + \mathcal{O}(log_2\ n * k) + \mathcal{O}(log_2\ n * k) = 3 * \mathcal{O}(log_2\ n * k) = \mathcal{O}(log_2\ n * k)$

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

```
1: if d.raiz == NULL then                                   O(1)
2:     res ← true                                           O(1)
3: else
4:     res ← false                                          O(1)
5: end if
```

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

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

```
1: var pNodo: puntero(nodo) ← d.raiz                        O(1)
2: while *(pNodo).der != NULL do                            O(log_2 n)
3:     pNodo ← *(pNodo).der                                 O(1)
4: end while
5: res ← *(pNodo).clave                                     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 ← CrearIt(d)                                         O(1)
```

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

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

```
1: res ← tupla(1, 0, d.tam, d.raiz, d.raiz)                 O(1)
```

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

IHayMas?(**in** it: `itClave`) $\rightarrow$ res : `bool`

```
1: if it.#nodosRecorridos, < it.tam - 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`) $\rightarrow$ res : α

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

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

```
IAvanzar(in/out it: itClave)
 1: it.#nodosRecorridos ← it.#nodosRecorridos + 1                                          O(1)
 2: var itNodosNivelActual: itClave ← crearIt(DameNodos(it.raiz, 1, it.nivelActual))       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.nodoActual ← Siguiente(itNodosNivelActual)                                      O(1)
13: else
14:     it.nivelActual ← it.nivelActual + 1                                                O(1)
15:     it.nodoActual ← Siguiente(crearIt(DameNodos(it.raiz, 1, it.nivelActual)))          O(n)
16: end if
```

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

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: $\mathcal{O}(n)$

Siendo n la cantidad de nodos. Vamos a ignorar las asignaciones y condicionales, porque tienen complejidad constante. Luego para justificar la complejidad de la recursión, se podría utilizar el Teorema Maestro, si fuese que el árbol está completamente lleno. De esta forma, siempre se divide en dos subproblemas y siempre se hace recursión sobre esos dos subproblemas. Aplicando el teorema maestro se tiene que: se divide en dos subproblemas (c = 2), se hace recursión sobre dos subproblemas (a = 2) y el costo de las funciones auxiliares es θ(n) (f(n) = θ(n)), y luego se tiene que $log_c\ a = log_2\ 2 = 1$). Finalmente se tiene que se cumple el segundo caso del Teorema Maestro, porque θ(n) $\in \theta(n^1)$, entonces la complejidad es $\theta(nlog_2\ n)$.

```
IRotar(in/out p: puntero(nodo))
 1: if FactorDesbalance(p) < -1 then                                                       O(1)
 2:     if FactorDesbalance(*(p).der) > 0 then                                             O(1)
 3:         res ← RotarDobleIzq(p)                                                         O(k)
 4:     else
 5:         res ← RotarSimpleIzq(p)                                                        O(k)
 6:     end if
 7: else
 8:     if FactorDesbalance(*(p).izq) < 0 then                                             O(1)
```

```
9:          res ←RotarDobleDer(p)                                   O(k)
10:     else
11:         res ← RotarSimpleDer(p)                                 O(k)
12:     end if
13: end if
```

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

Siendo k el costo de comparación de α

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                                            O(1)
8:     if r == *(padre).izq then                                    O(k)
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                                              O(1)
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: $\mathcal{O}(k)$

Siendo k el costo de comparación de α

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                                            O(1)
8:     if r == *(padre).izq then                                    O(k)
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                                                 O(1)
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: $\mathcal{O}(k)$

Siendo k el costo de comparación de α

```
IRotarDobleIzq(in/out p: puntero(nodo))
1: RotarSimpleDer(*(p).der)                                            O(k)
2: RotarSimpleIzq(p)                                                   O(k)
```

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

Siendo k el costo de comparación de α

```
IRotarDobleDer(in/out p: puntero(nodo))
1: RotarSimpleIzq(*(p).izq)                                            O(k)
2: RotarSimpleDer(p)                                                   O(k)
```

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

Siendo k el costo de comparación de α

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

ISiguientes(**in** *it* : `itClave`) → *res* : `Lista Enlazada`

```
1: var itVar: itClave ← it                                                 O(1)
2: var siguientes: Lista Enlazada ← Vacia()                                O(1)
3: while HayMas?(itVar) do                                                 O(it.tam^2)
4:     AgregarAtras(siguientes, Actual(itVar))                             O(1)
5:     Avanzar(it)                                                         O(it.tam)
6: end while
7: res ← siguientes                                                        O(1)
```

Complejidad: $\mathcal{O}(it.tam^2)$

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: $\mathcal{O}(1)$