

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	3
1.1. Secu(α)	3
1.2. Mapa	3
1.3. Diccionario(Clave, Significado)	3
1.4. Conjunto($\alpha <$)	3
2. Red	5
2.1. Interfaz	5
2.2. Auxiliares	6
2.3. Representacion	6
2.4. InvRep y Abs	6
2.5. Algoritmos	7
3. DCNet	11
3.1. Interfaz	11
3.2. Representacion	12
3.3. InvRep y Abs	12
3.4. Algoritmos	13
4. Diccionario String	17
4.1. Interfaz	17
5. Diccionario Rápido	18
5.1. Interfaz	18
5.2. Auxiliares	19
5.3. Representación	20
5.4. InvRep y Abs	21
5.5. Algoritmos	22
6. Extensión de Lista Enlazada(α)	31
6.1. Interfaz	31
6.2. Algoritmos	31
7. Extensión de Conjunto Lineal(α)	32
7.1. Interfaz	32
7.2. Algoritmos	32

1. Tad Extendidos

1.1. Secu(α)

otras operaciones

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

axiomas

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

1.2. Mapa

observadores básicos

restricciones : Mapa $m \longrightarrow \text{secu}(\text{restriccion})$

nroConexion : estacion $e_1 \times \text{estacion } e_2 \times \text{Mapa } m \longrightarrow \text{nat}\{e_1, e_2 \subset \text{estaciones}(m) \wedge_L \text{conectadas?}(e_1, e_2, m)\}$

axiomas

restricciones(vacio) $\equiv \langle \rangle$

restricciones(agregar(e, m)) $\equiv \text{restricciones}(m)$

restricciones(conectar(e_1, e_2, r, m)) $\equiv \text{restricciones}(m) \circ r$

nroConexion($e_1, e_2, \text{conectar}(e_3, e_4, m)$) \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($e_1, e_2, \text{agregar}(e, m)$) $\equiv \text{nroConexion}(e_1, e_2, m)$

1.3. Diccionario(Clave, Significado)

otras operaciones

vacío? : Diccionario $\longrightarrow \text{Bool}$

claveMax : Diccionario $d \longrightarrow \text{Clave}$

secuClaves: Diccionario $\longrightarrow \text{Secu}(\text{clave})$

$\{\neg \text{vacío}(d)\}$

axiomas

vacío?(vacío) $\equiv \text{true}$

vacío?(definir(c, s, d)) $\equiv \text{false}$

claveMax(d) $\equiv \text{elemMax}(\text{claves}(d))$

secuClaves(vacío) $\equiv \langle \rangle$

secuClaves(definir(c, s, d)) $\equiv \text{secuClaves}(d) \circ c$

1.4. Conjunto($\alpha <$)

otras operaciones

elemMax : Conj(α) $c \longrightarrow \alpha$

auxElemMax : $\alpha \times \text{Conj}(\alpha) \longrightarrow \alpha$

$\{\neg \emptyset?(c)\}$

axiomas

elemMax(c) $\equiv \text{auxMaxElem}(\text{dameUno}(c), c)$

```

auxElemMax(e, c)  ≡  if  $\emptyset?(c)$  then
    e
  else
    if  $e > \text{dameUno}(c)$  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 $\equiv \{\text{true}\}$

Post $\equiv \{res =_{\text{obs}} \text{crearIt}(\text{computadoras}(r))\}$

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 $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$

Post $\equiv \{res =_{\text{obs}} \text{conectadas?}(r, c_1, c_2)\}$

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 $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r) \wedge_L \text{conectadas?}(r, c_1, c_2)\}$

Post $\equiv \{res =_{\text{obs}} \text{interfazUsada}(r, c_1, c_2)\}$

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

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

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

Descripción: Crea una red sin computadoras.

AGREGARCOMPUTADORA(**in/out** r : red, **in** c : compu)

Pre $\equiv \{r_0 =_{\text{obs}} r \wedge \neg(c \in \text{computadoras}(r))\}$

Post $\equiv \{r =_{\text{obs}} \text{agregarComputadora}(r_0, c)\}$

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 $\equiv \{r_0 =_{\text{obs}} r \wedge \{c_1, c_2\} \subseteq \text{computadoras}(r) \wedge \text{ip}(c_1) \neq \text{ip}(c_2) \wedge_L \neg \text{conectadas?}(r, c_1, c_2) \wedge \neg \text{usaInterfaz?}(r, c_1, i_1) \wedge \neg \text{usaInterfaz?}(r, c_2, i_2)\}$

Post $\equiv \{r =_{\text{obs}} \text{conectar}(r, c_1, i_1, c_2, i_2)\}$

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 $\equiv \{c \in \text{computadoras}(r)\}$

Post $\equiv \{res =_{\text{obs}} \text{vecinos}(r, c)\}$

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 $\equiv \{c \in \text{computadoras}(r)\}$

Post $\equiv \{res =_{\text{obs}} \text{usaInterfaz?}(r, c, i)\}$

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 $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$

Post $\equiv \{res =_{\text{obs}} \text{crearItBi}(\text{caminosMinimos}(r, c_1, c_2))\}$

Descripción: Devuelve todos los caminos minimos de conexiones entre una computadora y otra

HAYCAMINO?(**in** $r : \text{red}$, **in** $c_1 : \text{compu}$, **in** $c_2 : \text{compu}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$

Post $\equiv \{res =_{\text{obs}} \text{hayCamino?}(r, c_1, c_2)\}$

Descripción: Verifica que haya un camino de conexiones entre una computadora y otra

2.2. Auxiliares

Operaciones auxiliares

CALCULARCAMINOSMINIMOS(**in** $r : \text{red}$, **in** $c_1 : \text{compu}$, **in** $c_2 : \text{compu}$) $\rightarrow res : \text{conj}(\text{lista})$

Pre $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$

Post $\equiv \{res =_{\text{obs}} \text{caminosMinimos}(r, c_1, c_2)\}$

Complejidad: $\mathcal{O}(\text{ALGO})$

Descripción: Devuelve los caminos minimos entre c_1 y c_2

CAMINOSIMPORTANTES(**in** $r : \text{red}$, **in** $c_1 : \text{compu}$, **in** $c_2 : \text{compu}$, **in** $\text{parcial} : \text{lista}$) $\rightarrow res : \text{conj}(\text{lista})$

Pre $\equiv \{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$

Post $\equiv \{res =_{\text{obs}} \text{caminosMinimos}(r, c_1, c_2)\}$

Complejidad: $\mathcal{O}(\text{ALGO})$

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), computasDirectas: conj(compu))), deOrigenADestino: diccString(compu: string, destinos: diccString(compu: string, caminosMinimos: conj(lista(compu)))) , computadoras: conj(compu))`

2.4. InvRep y Abs

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

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

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

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)

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

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

INTERFAZUSADA(in r : red, in c_1 : compu, in c_2 : compu) $\rightarrow res$: interfaz

1: $res \leftarrow \text{Obtener}(\text{Obtener}(r.directasEInterfaces, c_1.ip).directas, c_2.ip)$

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

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

INICIARRED() $\rightarrow res$: red

1: $res \leftarrow \text{tupla}(\text{directasEInterfaces: Vacio}(), \text{deOrigenADestino: Vacio}(), \text{computadoras: Vacio}())$ $\mathcal{O}(1+1+1)$

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

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

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

AGREGARCOMPUTADORA(in/out r : red, in c : compu)

1: Definir($r.directasEInterfaces, c.ip, \text{tupla}(\text{Vacio}(), \text{Vacio}())$)

$\mathcal{O}(|c.ip|)$

2: var $itComputadoras:itConj(\alpha) \leftarrow \text{CrearIt}(r.computadoras)$

$\mathcal{O}(1)$

3: while HaySiguiente?($itComputadoras$) do

$\mathcal{O}(1)$

4: Definir($\text{Obtener}(r.deOrigenADestino, \text{Siguiente}(itComputadoras).ip), c.ip, \text{Vacio}()$) $\mathcal{O}(|\text{Siguiente}(itComputadoras).ip| + |c.ip|)$

5: Avanzar($itComputadoras$)

$\mathcal{O}(1)$

6: end while

7: var $dicNuevaCompu:dicString(\alpha) \leftarrow \text{Vacio}()$

$\mathcal{O}(1)$

8: while HayAnterior?($itComputadoras$) do

$\mathcal{O}(1)$

9: Definir($dicNuevaCompu, \text{Anterior}(itComputadoras).ip, \text{Vacio}()$)

$\mathcal{O}(1)$

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

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

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

CONECTAR(in/out r : red, in c_1 : compu, in i_1 : interfaz, in c_2 : compu, in i_2 : interfaz)

1: var $tupSig1:tupla \leftarrow \text{Obtener}(r.directasEInterfaces, c_1.ip)$

2: Definir($tupSig1.directas, c_2.ip, i_1$)

$\mathcal{O}(|c_1| + |c_2| + 1)$

3: Agregar($tupSig1.compDirectas, c_2$)

$\mathcal{O}(1)$

4: var $tupSig2:tupla \leftarrow \text{Obtener}(r.directasEInterfaces, c_2.ip)$

5: Definir($tupSig2.directas, c_1.ip, i_2$)

$\mathcal{O}(|c_1| + |c_2| + 1)$

6: Agregar($tupSig2.compDirectas, c_1$)

$\mathcal{O}(1)$

7: var $itOrigenes:itConj(\alpha) \leftarrow \text{CrearIt}(r.computadoras)$

$\mathcal{O}(1)$

8: while HaySiguiente?($itOrigenes$) do

$\mathcal{O}(1)$

9: var $dicCompu:dicString(\alpha) \leftarrow \text{Obtener}(r.deOrigenADestino, \text{Siguiente}(itOrigenes).ip)$

$\mathcal{O}(1)$

10: var $itDestinos:itConj(\alpha) \leftarrow \text{CrearIt}(r.computadoras)$

$\mathcal{O}(1)$

11: while HaySiguiente?($itDestinos$) do

$\mathcal{O}(1)$

12: if $\text{Siguiente}(itOrigenes) \neq \text{Siguiente}(itDestinos)$ then

$\mathcal{O}(1)$

13: Definir($dicCompu, \text{Siguiente}(itDestinos).ip, \text{CalcularCaminosMinimos}(r, \text{Siguiente}(itOrigenes), \text{Siguiente}(itDestinos))$)

$\mathcal{O}(1)$

14: end if

15: Avanzar($itDestinos$)

$\mathcal{O}(1)$

16: end while

17: Avanzar($itOrigenes$)

$\mathcal{O}(1)$

18: end while

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

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

```

IVECINOS(in r : red, in c : compu) → res : conj(compu)
1: res ← 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 c1 : compu, in c2 : compu) → res : itConj(α)
1: res ← CrearIt(Obtener(Obtener(r.deOrigenADestino, c1.ip), c2.ip)) O(1)

```

Complejidad:

```

IHAYCAMINO(in r : red, in c1 : compu, in c2 : compu) → res : bool
1: var conjCaminosMinimos ← CaminosMinimos(r, c1, c2)           O(1)
2: res ← EsVacio?(conjCaminosMinimos)                             O(1)

```

Complejidad:

```

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

```

Complejidad:

```
ICAMINOSIMPORTANTES(in r : red, in c1 : compu, in c2 : compu, in pacial : lista(compu) → res : conj(lista)
1: res ← Vacio()  $\mathcal{O}(1)$ 
2: if Pertenece?(Vecinos(r, c1), c2) then  $\mathcal{O}(1)$ 
3:   AgregarAtras(pacial, c2)  $\mathcal{O}(1)$ 
4:   Agregar(res, pacial)  $\mathcal{O}(1)$ 
5: else
6:   var itVecinos:itConj ← CrearIt(Vecinos(r, c1))  $\mathcal{O}(1)$ 
7:   while HaySiguiente?(itVecinos) do  $\mathcal{O}(1)$ 
8:     if ¬Pertenece?(pacial, Siguiente(itVecinos)) then  $\mathcal{O}(1)$ 
9:       var auxParcial:lista ← pacial  $\mathcal{O}(1)$ 
10:      AgregarAtras(auxParcial, Siguiente(itVecinos))  $\mathcal{O}(1)$ 
11:      Unir(res, CaminosImportantes(r, Siguiente(itVecinos), c2, auxParcial))  $\mathcal{O}(1)$ 
12:    end if
13:    Avanzar(itVecinos)  $\mathcal{O}(1)$ 
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 $\equiv \{res =_{obs} red(d)\}$

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

Descripción: Devuelve la red del dcnet.

CAMINORECORRIDO(in d : dcnet, in p : paquete) $\rightarrow res$: lista(compu)

Pre $\equiv \{p \in paqueteEnTransito?(d, p)\}$

Post $\equiv \{res =_{obs} caminoRecorrido(d, p)\}$

Complejidad: $\mathcal{O}(n * \log_2(k))$, donde $n = \#computadoras(red(d))$, y k la cantidad maxima de paquetes que hay en una computadora.

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 $\equiv \{res =_{obs} cantidadEnviados(d, c)\}$

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

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 $\equiv \{res =_{obs} enEspera(d, c)\}$

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

Descripción: Devuelve los paquetes que se encuentran en ese momento en la computadora.

INICIARDCNET(in r : red) $\rightarrow res$: dcnet

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} iniciarDCNet(r)\}$

Complejidad: $\mathcal{O}(n * L)$, donde $n = \#computadoras(red(d))$, y L es el hostname mas largo de las computadoras.

Descripción: Inicia un dcnet con la red y sin paquetes.

CREARPAQUETE(in p : paquete, in/out d : dcnet)

Pre $\equiv \{d_0 \equiv d \wedge \neg ((\exists p_1: paquete)(paqueteEnTransito(s, p_1) \wedge id(p_1) = id(p)) \wedge origen(p) \in computadoras(red(d)) \wedge_L destino(p) \in computadoras(red(d)) \wedge_L hayCamino?(red(d, origen(p), destino(p))) \}$

Post $\equiv \{d =_{obs} crearPaquete(d_0)\}$

Complejidad: $\mathcal{O}(L + \log_2(k))$, donde L es el hostname mas largo de las computadoras y k la cantidad maxima de paquetes que hay en una computadora.

Descripción: Agrega el paquete al dcnet.

AVANZARSEGUNDO(in/out d : dcnet)

Pre $\equiv \{d_0 \equiv d\}$

Post $\equiv \{d =_{obs} avanzarSegundo(d_0)\}$

Complejidad: $\mathcal{O}(n * (L + \log_2(k)))$, donde $n = \#computadoras(red(d))$ y k la cantidad maxima de paquetes que hay en una computadora.

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 $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} \text{paqueteEnTransito?}(d,p)\}$

Complejidad: $\mathcal{O}(n * \log_2(k))$, donde $n = \#computadoras(\text{red}(d))$ y k la cantidad maxima de paquetes que hay en una computadora.

Descripción: Devuelve si el paquete esta o no en alguna computadora del sistema.

LAQUEMASENVIO(in $d: \text{dcnet}$) $\rightarrow res: \text{compu}$

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

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

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 lista enlazada de `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(ip: string, tupla(MasPriori:DiccRapido(prioridad :nat,`
`PaqdePriori:ConjLineal(paquete)), PaqYCam:DiccRapido (paq:paquete, CamRecorri-`
`do:ListaEnlazada(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`. Entonces, `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`

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

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

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

$\text{Abs} : e_dc \ e \longrightarrow \text{dcnet} \quad \{\text{Rep}(e)\}$
 $\text{Abs}(e) =_{\text{obs}} d : \text{dcnet} \mid$
 $\text{red}(d) = e.\text{red} \wedge_L$
 $(\forall p:\text{paquete}, c:\text{compu})(c \in \text{computadoras}(\text{red}(d)) \wedge (\text{paqueteEnTransito?}(d, p) \Rightarrow_L (\text{def?}(c.\text{ip},$
 $e.\text{CompYPaq} \wedge_L \text{def?}(p, \text{obtener}(c.\text{ip}, e.\text{CompYPaq}).\text{PaqYCam}) \Rightarrow_L \text{obtener}(p, \text{obtener}(c.\text{ip},$
 $e.\text{CompYPaq}).\text{PaqYCam}) = \text{caminoRecorrido}(d, p)))$
 $(\forall c:\text{compu}) (c \in \text{computadoras}(\text{red}(d)) \Rightarrow_L ((\text{def?}(c.\text{ip}, e.\text{CompYPaq}) \Rightarrow_L \text{obtener}(c.\text{ip},$
 $e.\text{CompYPaq}).\text{Enviados} = \text{cantidadEnviados}(d, c)) \wedge (\text{def?}(c.\text{ip}, e.\text{CompYPaq}) \Rightarrow_L \text{cla-}$
 $\text{ves}(\text{obtener}(c.\text{ip}, e.\text{CompYPaq}).\text{PaqYCam}) = \text{enEspera}(d, c))))$

3.4. Algoritmos

Algoritmos

IREDA(**in** $d : \text{dcnet}$) $\rightarrow res : \text{red}$

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

$\mathcal{O}(1)$

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

ICAMINORECORRIDO(in d : dcnet, in p : paquete) $\rightarrow res$: lista(compu)

1: var $it \leftarrow \text{CREAIT}(d.\text{CompYPaq})$	$\mathcal{O}(n)$
2: var $esta$: bool $\leftarrow \text{false}$	$\mathcal{O}(1)$
3: while HAYSIGUIENTE(it) $\wedge \neg esta$ do	$\mathcal{O}(n)$
4: var $diccpaq$: diccRapido $\leftarrow ((\text{SIGUIENTE}(it)).\text{significado}).\text{PaqYCam}$	$\mathcal{O}(1)$
5: if DEF?($p, diccpaq$) then	$\mathcal{O}(\log_2(k))$
6: $esta \leftarrow \text{true}$	$\mathcal{O}(1)$
7: $res \leftarrow \text{OBTENER}(p, diccpaq)$	$\mathcal{O}(\log_2(k))$
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.

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

ICANTIDADENVIADOS(in d : dcnet, in c : compu) $\rightarrow res$: nat

1: $res \leftarrow \text{OBTENER}(c.\text{ip}, d.\text{CompYPaq}).\text{Enviados}$	$\mathcal{O}(L)$
--	------------------

Complejidad: $\mathcal{O}(L)$ Siendo L la longitud de el IP de c

IENESPERA(in d : dcnet, in c : compu) $\rightarrow res$: itPaquete)

1: $res \leftarrow \text{CLAVES}(\text{OBTENER}(c.\text{ip}, d.\text{CompYPaq}).\text{PaqYCam})$	$\mathcal{O}(L)$
--	------------------

Complejidad: $\mathcal{O}(L)$ Siendo L la longitud del IP de c

IINICIARDCNET(in r : red, in/out d : dcnet)

1: $d.\text{red} \leftarrow r$	$\mathcal{O}(1)$
2: var $it \leftarrow \text{COMPUTADORAS}(\text{red})$	$\mathcal{O}(1)$
3: $d.\text{MasEnviante} \leftarrow \text{tupla}(\text{SIGUIENTE}(it), 0)$	$\mathcal{O}(1)$
4: $d.\text{CompyPaq} \leftarrow \text{Vacio}()$	$\mathcal{O}(1)$
5: while HAYSIGUIENTE(it) do	$\mathcal{O}(n)$
6: DEFINIR($\text{SIGUIENTE}(it).\text{ip}$, $\text{tupla}(\text{VACIO}(), \text{VACIO}(), 0)$, $d.\text{CompyPaq}$)	$\mathcal{O}(L + 1 + 1)$
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 IP mas largo de ellas.

$$\begin{aligned} &\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) + n * \mathcal{O}(L + 1 + 1) + \mathcal{O}(1) = \\ &\mathcal{O}(n * L) \end{aligned}$$

ICREARPAQUETE(in p : paquete, in/out d : dcnet)

1: var $diccprio$: diccRapido $\leftarrow \text{OBTENER}(p.\text{origen}, d.\text{CompYPaq}).\text{MasPriori}$	$\mathcal{O}(L)$
2: var $dicccam$: diccRapido $\leftarrow \text{OBTENER}(p.\text{origen}, d.\text{CompYPaq}).\text{PaqYCam}$	$\mathcal{O}(L)$
3: if $\neg \text{DEF?}(p.\text{prioridad}, diccprio)$ then	$\mathcal{O}(\log_2(k))$
4: var cj :conjunto $\leftarrow \text{VACIO}()$	$\mathcal{O}(1)$
5: AGREGAR(cj, p)	$\mathcal{O}(1)$
6: DEFINIR($p.\text{prioridad}, cj, diccprio$)	$\mathcal{O}(\log_2(k))$
7: else	
8: var cj :conjunto $\leftarrow \text{OBTENER}(p.\text{prioridad}, diccprio)$	$\mathcal{O}(\log_2(k))$
9: AGREGAR(cj, p)	$\mathcal{O}(1)$

10: DEFINIR(<i>p.prioridad</i> , <i>cj</i> , <i>diccprio</i>)	$\mathcal{O}(\log_2(k))$
11: end if	
12: var <i>l:lista</i> \leftarrow VACIA()	$\mathcal{O}(1)$
13: AGREGARATRÁS(<i>l</i> , <i>p.origen</i>)	$\mathcal{O}(1)$
14: DEFINIR(<i>p</i> , <i>l</i> , <i>dicccam</i>)	$\mathcal{O}(\log_2(k))$

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.

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

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

1: var <i>it</i> \leftarrow COMPUTADORAS(<i>red</i>)	$\mathcal{O}(1)$
2: var <i>aux</i> \leftarrow VACIA()	$\mathcal{O}(1)$
3: while HAYSIGUIENTE(<i>it</i>) do	$\mathcal{O}(n)$
4: var <i>diccprio</i> : <i>diccRapido</i> \leftarrow OBTENER(SIGUIENTE(<i>it</i>).ip, <i>d.CompYPaq</i>).MasPriori)	$\mathcal{O}(L)$
5: var <i>dicccam</i> : <i>diccRapido</i> \leftarrow OBTENER(SIGUIENTE(<i>it</i>).ip, <i>d.CompYPaq</i>).PaqYCam)	$\mathcal{O}(L)$
6: if \neg VACIO?(<i>diccprio</i>) then	$\mathcal{O}(1)$
7: var <i>paq</i> : paquete \leftarrow PRIMERO(OBTENER(CLAVEMAX(<i>diccprio</i>), <i>diccprio</i>))	$\mathcal{O}(\log_2(k) + 1 + 1)$
8: AGREGARADELANTE(<i>aux</i> , tupla(<i>paq</i> : <i>paq</i> , <i>pcant</i> : <i>it</i> .ip, <i>camrecorrido</i> : OBTENER(<i>paq</i> , <i>dicccam</i>)))	$\mathcal{O}(1 + \log_2(k))$
9: ELIMINAR(OBTENER(CLAVEMAX(<i>diccprio</i>), <i>diccprio</i>), <i>paq</i>)	$\mathcal{O}(\log_2(k) + \log_2(k) + 1)$
10: if ESVACIO?(OBTENER(CLAVEMAX(<i>diccprio</i>), <i>diccprio</i>) then	$\mathcal{O}(\log_2(k))$
11: BORRAR(CLAVEMAX(<i>diccprio</i>), <i>diccprio</i>)	$\mathcal{O}(\log_2(k))$
12: end if	
13: BORRAR(<i>paq</i> , <i>dicccam</i>)	$\mathcal{O}(\log_2(k))$
14: OBTENER(SIGUIENTE(<i>it</i>).ip, <i>d.CompYPaq</i>).Enviados ++	$\mathcal{O}(L)$
15: if OBTENER(SIGUIENTE(<i>it</i>).ip, <i>d.CompYPaq</i>).Enviados > (<i>d.MasEnviante</i>).enviados then	$\mathcal{O}(L + 1)$
16: <i>d.MasEnviante</i> \leftarrow tupla(SIGUIENTE(<i>it</i>), OBTENER(SIGUIENTE(<i>it</i>).ip, <i>d.CompYPaq</i>).Enviados)	$\mathcal{O}(L + 1)$
17: end if	
18: end if	
19: AVANZAR(<i>it</i>)	$\mathcal{O}(1)$
20: end while	
21: var <i>itaux</i> \leftarrow CREAMIT(<i>aux</i>)	$\mathcal{O}(1)$
22: while HAYSIGUIENTE(<i>itaux</i>) do	$\mathcal{O}(n)$
23: var <i>proxpc</i> : compu \leftarrow PRIMERO(SIGUIENTE(CAMINOSMINIMOS(<i>d.red</i> , SIGUIENTE(<i>itaux</i>).pcant, SIGUIENTE(<i>itaux</i>).destino))	$\mathcal{O}(L + L)$
24: var <i>diccprio</i> : <i>diccRapido</i> \leftarrow OBTENER(<i>proxpc</i> .ip, <i>d.CompYPaq</i>).MasPriori)	$\mathcal{O}(L)$
25: var <i>dicccam</i> : <i>diccRapido</i> \leftarrow OBTENER(<i>proxpc</i> .ip, <i>d.CompYPaq</i>).PaqYCam)	$\mathcal{O}(L)$
26: if <i>proxpc</i> \neq (SIGUIENTE(<i>itaux</i>).paq).destino then	
27: if DEF?((SIGUIENTE(<i>itaux</i>).paq).prioridad, <i>diccprio</i>) then	$\mathcal{O}(\log_2(k))$
28: var <i>mismaprio</i> : conj(paquetes) \leftarrow AGREGAR(OBTENER(SIGUIENTE(<i>itaux</i>).paq.prioridad, <i>diccprio</i>), SIGUIENTE(<i>itaux</i>).paq)	$\mathcal{O}(\log_2(k))$
29: DEFINIR((SIGUIENTE(<i>itaux</i>).paq).prioridad, <i>mismaprio</i> , <i>diccprio</i>)	$\mathcal{O}(\log_2(k))$
30: else	
31: DEFINIR(SIGUIENTE(<i>itaux</i>).prioridad, AGREGAR(VACIO(),SIGUIENTE(<i>itaux</i>).paq), <i>diccprio</i>)	$\mathcal{O}(\log_2(k))$
32: end if	
33: DEFINIR(<i>p</i> .paq,AGREGARATRÁS(SIGUIENTE(<i>itaux</i>).camrecorrido , <i>proxpc</i>), <i>dicccam</i>)	$\mathcal{O}(\log_2(k))$
34: end if	
35: AVANZAR(SIGUIENTE(<i>itaux</i>))	$\mathcal{O}(1)$
36: 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.

$$\begin{aligned}
& \mathcal{O}(1) + \mathcal{O}(1) + n * (\mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(1) + \mathcal{O}(\log_2(k) + 1 + 1) + \mathcal{O}(1 + \log_2(k)) + \mathcal{O}(\log_2(k) + \log_2(k) + 1) + \\
& \mathcal{O}(\log_2(k)) + \mathcal{O}(\log_2(k)) + \mathcal{O}(\log_2(k)) + \mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(1)) + n * (\mathcal{O}(L) + \mathcal{O}(L) + \mathcal{O}(\log_2(k)) + \\
& \mathcal{O}(\log_2(k)) + \mathcal{O}(\log_2(k)) + \mathcal{O}(\log_2(k))\mathcal{O}(\log_2(k)) + \mathcal{O}(1) + \mathcal{O}(1)) = \\
& n * (\mathcal{O}(5 * L + 5 * \log_2(k))) + n * \mathcal{O}(3 * L + 5 * \log_2(k)) = \\
& 2n * (\mathcal{O}(L + \log_2(k))) = \\
& \mathcal{O}(n * (L + \log_2(k)))
\end{aligned}$$

IPAQUETEENTRANSITO?(**in** *d*: **dcnet**, **in** *p*: **paquete**) \rightarrow *res* : **bool**

1: var <i>it</i> \leftarrow CREAMIT(COMPUTADORAS(<i>d</i> .red)	$\mathcal{O}(1)$
2: var <i>esta</i> : bool \leftarrow false	$\mathcal{O}(1)$
3: while HAYSIGUIENTE(<i>it</i>) \wedge \neg <i>esta</i> do	$\mathcal{O}(n)$
4: <i>esta</i> \leftarrow DEF?(OBTENER(<i>d</i> .CompYPaq,SIGUIENTE(<i>i</i>).id).PqYCam , <i>p</i>)	$\mathcal{O}(\log_2(k))$
5: AVANZAR(<i>it</i>)	$\mathcal{O}(1)$
6: end while	
7: <i>res</i> \leftarrow <i>esta</i>	$\mathcal{O}(1)$

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.

$$\begin{aligned}
& \mathcal{O}(1) + \mathcal{O}(1) + n * (\mathcal{O}(\log_2(k)) + \mathcal{O}(1)) + \mathcal{O}(1) = \\
& \mathcal{O}(3) + n * (\mathcal{O}(\log_2(k))) = \\
& \mathcal{O}(n * \log_2(k))
\end{aligned}$$

ILAQUEMASENVIO(**in** *d*: **dcnet**) \rightarrow *res* : **compu**

1: <i>res</i> \leftarrow (<i>d</i> .MasEnviante).compu	$\mathcal{O}(1)$
---	------------------

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

Post $\equiv \{res =_{\text{obs}} \text{def?}(c, d)\}$

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

Descripción: Revisa si la clave ingresada se encuentra definida en el diccionario

VACÍO() $\rightarrow res$: diccString(string, β)

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

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

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

Descripción: Crea nuevo diccionario vacío

OBTENER(in c : string, in d : diccString(string, β)) $\rightarrow res$: β

Pre $\equiv \{\text{def?}(c, d)\}$

Post $\equiv \{res =_{\text{obs}} \text{obtener}(c, d)\}$

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 $\equiv \{d =_{\text{obs}} \text{definir}(c, s, d_0)\}$

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 $\equiv \{d =_{\text{obs}} d_0 \wedge \text{def?}(c, d_0)\}$

Post $\equiv \{res =_{\text{obs}} \text{borrar}(c, d_0)\}$

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: $\text{diccRapido}(\alpha, \beta)$, $\text{ITCLAVE}(\alpha)$

Operaciones básicas de Diccionario Rápido(α, β)

DEF?(**in** $c : \alpha$, **in** $d : \text{diccRapido}(\alpha, \beta)$) $\rightarrow res : \text{bool}$

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

Post $\equiv \{res =_{\text{obs}} \text{def?}(c, d)\}$

Complejidad: $\mathcal{O}(\log_2 n)$, siendo n la cantidad de claves

Descripción: Verifica si una clave está definida.

OBTENER(**in** $c : \alpha$, **in** $d : \text{diccRapido}(\alpha, \beta)$) $\rightarrow res : \beta$

Pre $\equiv \{\text{def?}(c, d)\}$

Post $\equiv \{res =_{\text{obs}} \text{obtener}(c, d)\}$

Complejidad: $\mathcal{O}(\log_2 n)$, siendo n la cantidad de claves

Descripción: Devuelve el significado asociado a una clave

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

VACÍO() $\rightarrow res : \text{diccRapido}(\alpha, \beta)$

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

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

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

Descripción: Crea un nuevo diccionario vacío

DEFINIR(**in** $c : \alpha$, **in** $s : \beta$, **in/out** $d : \text{diccRapido}(\alpha, \beta)$)

Pre $\equiv \{d =_{\text{obs}} d_0\}$

Post $\equiv \{d =_{\text{obs}} \text{definir}(c, s, d_0)\}$

Complejidad: $\mathcal{O}(\log_2 n)$, siendo n la cantidad de claves

Descripción: Define la clave, asociando su significado, al diccionario

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

BORRAR(**in** $c : \alpha$, **in/out** $d : \text{diccRapido}(\alpha, \beta)$)

Pre $\equiv \{d =_{\text{obs}} d_0 \wedge \text{def?}(c, d_0)\}$

Post $\equiv \{d =_{\text{obs}} \text{borrar}(c, d_0)\}$

Complejidad: $\mathcal{O}(\log_2 n)$, siendo n la cantidad de claves

Descripción: Borra la clave (y su significado) del diccionario

VACÍO?(**in** $d : \text{diccRapido}(\alpha, \beta)$) $\rightarrow res : \text{bool}$

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

Post $\equiv \{res =_{\text{obs}} \text{vacío?}(d)\}$

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

Descripción: Verifica si el diccionario vacío

CLAVEMAX(**in** $d : \text{diccRapido}(\alpha, \beta)$) $\rightarrow res : \alpha$

Pre $\equiv \{\neg \text{vacío?}(d)\}$

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

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

Descripción: Devuelve la mayor clave

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

CLAVES(**in** $d : \text{diccRapido}(\alpha, \beta)$) $\rightarrow res : \text{itClave}$

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

Post $\equiv \{res =_{\text{obs}} \text{CrearIt}(\text{claves}(d))\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un iterador de paquete

Operaciones del Iterador

CREARIT(**in** $d: \text{diccRapido}(\alpha, \beta) \rightarrow res: \text{itClave}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{crearItUni}(\text{secuClaves}(d))\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Crea el iterador de claves
Aliasing: Res es devuelto por referencia

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

ACTUAL(**in** $it: \text{itClave} \rightarrow res: \alpha$
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** $it: \text{itClave}$
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** $p: \text{puntero}(\text{nodo}), \text{in } actual: \text{nat}, \text{in } destino: \text{nat} \rightarrow res: \text{conjLineal}(\text{nodo})$
Pre $\equiv \{\text{true}\}$
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** $p: \text{puntero}(\text{nodo})$
Pre $\equiv \{p =_{\text{obs}} p_0 \wedge p \neq \text{NULL} \wedge_L (|\text{FACTORDESBALANCE}(p)| > 1)\}$
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** $p: \text{puntero}(\text{nodo})$
Pre $\equiv \{p =_{\text{obs}} p_0 \wedge p \neq \text{NULL} \wedge_L *(p).der \neq \text{NULL}\}$
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** $p: \text{puntero}(\text{nodo})$
Pre $\equiv \{p =_{\text{obs}} p_0 \wedge p \neq \text{NULL} \wedge_L *(p).izq \neq \text{NULL}\}$
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 $\equiv \{p =_{\text{obs}} p_0 \wedge p \neq \text{NULL} \wedge_L *(p).\text{der} \neq \text{NULL} \wedge_L (*(p).\text{der}).\text{izq} \neq \text{NULL}\}$

Post $\equiv \{p =_{\text{obs}} \text{rotarDobleIzq}(p_0)\}$

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

Descripción: Realiza una rotación doble izquierda del nodo p , y los nodos involucrados

ROTARDOBLEDER(**in/out** p : puntero(nodo))

Pre $\equiv \{p =_{\text{obs}} p_0 \wedge p \neq \text{NULL} \wedge_L *(p).\text{izq} \neq \text{NULL} \wedge_L (*(p).\text{izq}).\text{der} \neq \text{NULL}\}$

Post $\equiv \{p =_{\text{obs}} \text{rotarDobleDer}(p_0)\}$

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 $\equiv \{p \neq \text{NULL}\}$

Post $\equiv \{res =_{\text{obs}} \text{altura}(p)\}$

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

Descripción: Calcula y devuelve la altura actual de p

FACTORDESBALANCE(**in** p : puntero(nodo)) $\rightarrow res$: nat

Pre $\equiv \{p \neq \text{NULL}\}$

Post $\equiv \{res =_{\text{obs}} \text{factorDesbalance}(p)\}$

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

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

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.

$\text{diccRapido}(\alpha, \beta)$ se representa con **estr**

donde **estr** es $\text{tupla}(\text{raiz: puntero(nodo)}, \text{tam: nat})$

donde **nodo** es $\text{tupla}(\text{clave: } \alpha, \text{significado: } \beta, \text{padre: puntero(nodo)}, \text{izq: puntero(nodo)}, \text{der: puntero(nodo)}, \text{alt: nat})$

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 $\text{tupla}(\text{nivelActual: nat}, \text{\#nodosRecorridos: nat}, \text{tam: nat}, \text{nodoActual: puntero(nodo)}, \text{raiz: puntero(nodo)})$

donde **nodo** es $\text{tupla}(\text{clave: } \alpha, \text{significado: } \beta, \text{padre: puntero(nodo)}, \text{izq: puntero(nodo)}, \text{der: puntero(nodo)}, \text{alt: nat})$

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:

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

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.raíz 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 $\log_2 \text{it.tam} + 1$ (inclusive).
6. El invariante de nodo es el mismo que el invariante de nodo del Diccionario.

Abs:

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

5.5. Algoritmos

Algoritmos

```

IDEF?(in c:  $\alpha$ , in d: diccRapido( $\alpha, \beta$ ))  $\rightarrow$  res : bool
1: var pNodo: puntero(nodo)  $\leftarrow$  d.raiz  $\mathcal{O}(1)$ 
2: while pNodo != NULL do  $\mathcal{O}(\log_2 n * k)$ 
3:   if *(pNodo).clave == c then  $\mathcal{O}(k)$ 
4:     res  $\leftarrow$  true  $\mathcal{O}(1)$ 
5:     return res  $\mathcal{O}(1)$ 
6:   else
7:     if c > *(pNodo).clave then  $\mathcal{O}(k)$ 
8:       pNodo  $\leftarrow$  *(pNodo).der  $\mathcal{O}(1)$ 
9:     else
10:      pNodo  $\leftarrow$  *(pNodo).izq  $\mathcal{O}(1)$ 
11:    end if
12:  end if
13: end while
14: res  $\leftarrow$  false  $\mathcal{O}(1)$ 

```

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:  $\alpha$ , in d: diccRapido( $\alpha, \beta$ ))  $\rightarrow$  res :  $\beta$ 
1: var pNodo: puntero(nodo)  $\leftarrow$  d.raiz  $\mathcal{O}(1)$ 
2: while *(pNodo).clave != c do  $\mathcal{O}(\log_2 n * k)$ 
3:   if c > *(pNodo).clave then  $\mathcal{O}(k)$ 
4:     pNodo  $\leftarrow$  *(pNodo).der  $\mathcal{O}(1)$ 
5:   else
6:     pNodo  $\leftarrow$  *(pNodo).izq  $\mathcal{O}(1)$ 
7:   end if
8: end while
9: res  $\leftarrow$  *(pNodo).significado  $\mathcal{O}(1)$ 

```

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( $\alpha, \beta$ )
1: var res: diccRapido( $\alpha, \beta$ )  $\leftarrow$  tupla(NULL, 0)  $\mathcal{O}(1)$ 

```

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

IDEFINIR(in $c: \alpha$, in $s: \beta$, in/out $d: \text{diccRapido}(\alpha, \beta)$)

```
1:
2: if d.raiz == NULL then  $\mathcal{O}(1)$ 
3:   d.raiz  $\leftarrow$  &tupla(c, s, NULL, NULL, NULL, 1)  $\mathcal{O}(1)$ 
4:   d.tam  $\leftarrow$  1  $\mathcal{O}(1)$ 
5: else
6:   if DEF?(c, d) then  $\mathcal{O}(\log_2 n * k)$ 
7:     var pNodo: puntero(nodo)  $\leftarrow$  d.raiz  $\mathcal{O}(1)$ 
8:     while *(pNodo).clave != c do  $\mathcal{O}(\log_2 n * k)$ 
9:       if c > *(pNodo).clave then  $\mathcal{O}(k)$ 
10:        pNodo  $\leftarrow$  *(pNodo).der  $\mathcal{O}(1)$ 
11:       else
12:        pNodo  $\leftarrow$  *(pNodo).izq  $\mathcal{O}(1)$ 
13:       end if
14:     end while
15:     *(pNodo).significado  $\leftarrow$  s  $\mathcal{O}(1)$ 
16:   else
17:     var seguir: bool  $\leftarrow$  true  $\mathcal{O}(1)$ 
18:     var pNodo: puntero(nodo)  $\leftarrow$  d.raiz  $\mathcal{O}(1)$ 
19:     while seguir == true do  $\mathcal{O}(\log_2 n * k)$ 
20:       if c > *(pNodo).clave  $\wedge$  *(pNodo).der == NULL then  $\mathcal{O}(k)$ 
21:        *(pNodo).der  $\leftarrow$  &tupla(c, s, pNodo, NULL, NULL, 1)  $\mathcal{O}(1)$ 
22:        seguir  $\leftarrow$  false  $\mathcal{O}(1)$ 
23:       else
24:        if c > *(pNodo).clave  $\wedge$  *(pNodo).der != NULL then  $\mathcal{O}(k)$ 
25:         pNodo  $\leftarrow$  *(pNodo).der  $\mathcal{O}(1)$ 
26:        else
27:         if c < *(pNodo).clave  $\wedge$  *(pNodo).izq == NULL then  $\mathcal{O}(k)$ 
28:          *(pNodo).izq  $\leftarrow$  &tupla(c, s, pNodo, NULL, NULL, 1)  $\mathcal{O}(1)$ 
29:          seguir  $\leftarrow$  false  $\mathcal{O}(1)$ 
30:        else
31:         pNodo  $\leftarrow$  *(pNodo).izq  $\mathcal{O}(1)$ 
32:        end if
33:      end if
34:    end if
35:  end while
36:  d.tam  $\leftarrow$  d.tam + 1  $\mathcal{O}(1)$ 
37:  while pNodo != NULL do  $\mathcal{O}(\log_2 n * k)$ 
38:    var padrePNodo  $\leftarrow$  *(pNodo).padre  $\mathcal{O}(1)$ 
39:    if |FACTORDESBALANCE(pNodo)| > 1 then  $\mathcal{O}(1)$ 
40:      ROTAR(pNodo)  $\mathcal{O}(k)$ 
41:    else
42:      *(pNodo).alt  $\leftarrow$  ALTURA(pNodo)  $\mathcal{O}(1)$ 
43:    end if
44:    pNodo  $\leftarrow$  padrePNodo  $\mathcal{O}(1)$ 
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:  $\alpha$ , in/out d: diccRapido( $\alpha, \beta$ ))
1: var pNodo: puntero(nodo)  $\leftarrow$  d.raiz  $\mathcal{O}(1)$ 
2: while c != *(pNodo).clave do  $\mathcal{O}(\log_2 n * k)$ 
3:   if c > *(pNodo).clave then  $\mathcal{O}(k)$ 
4:     pNodo  $\leftarrow$  *(pNodo).der  $\mathcal{O}(1)$ 
5:   else
6:     pNodo  $\leftarrow$  *(pNodo).izq  $\mathcal{O}(1)$ 
7:   end if
8: end while
9: if *(pNodo).izq == NULL  $\wedge$  *(pNodo).der == NULL then  $\mathcal{O}(1)$ 
10:  if *(pNodo).padre == NULL then  $\mathcal{O}(1)$ 
11:    d.raiz  $\leftarrow$  NULL  $\mathcal{O}(1)$ 
12:  else
13:    if pNodo == (*(pNodo).padre).izq then  $\mathcal{O}(k)$ 
14:      (*(pNodo).padre).izq  $\leftarrow$  NULL  $\mathcal{O}(1)$ 
15:    else
16:      (*(pNodo).padre).der  $\leftarrow$  NULL  $\mathcal{O}(1)$ 
17:    end if
18:  end if
19: else
20:  if *(pNodo).izq == NULL  $\wedge$  *(pNodo).der != NULL then  $\mathcal{O}(1)$ 
21:    if *(pNodo).padre == NULL then  $\mathcal{O}(1)$ 
22:      (*(pNodo).der).padre  $\leftarrow$  NULL  $\mathcal{O}(1)$ 
23:      d.raiz  $\leftarrow$  *(pNodo).der  $\mathcal{O}(1)$ 
24:    else
25:      if pNodo == (*(pNodo).padre).izq then  $\mathcal{O}(k)$ 
26:        (*(pNodo).padre).izq  $\leftarrow$  *(pNodo).der  $\mathcal{O}(1)$ 
27:      else
28:        (*(pNodo).padre).der  $\leftarrow$  *(pNodo).der  $\mathcal{O}(1)$ 
29:      end if
30:      (*(pNodo).der).padre  $\leftarrow$  *(pNodo).padre  $\mathcal{O}(1)$ 
31:    end if
32:  else
33:    if *(pNodo).izq != NULL  $\wedge$  *(pNodo).der == NULL then  $\mathcal{O}(1)$ 
34:      if *(pNodo).padre == NULL then  $\mathcal{O}(1)$ 
35:        (*(pNodo).izq).padre  $\leftarrow$  NULL  $\mathcal{O}(1)$ 
36:        d.raiz  $\leftarrow$  *(pNodo).izq  $\mathcal{O}(1)$ 
37:      else
38:        if pNodo == (*(pNodo).padre).izq then  $\mathcal{O}(k)$ 

```


39:	$*(pNodo).padre).izq \leftarrow *(pNodo).izq$	$\mathcal{O}(1)$
40:	else	
41:	$*(pNodo).padre).der \leftarrow *(pNodo).izq$	$\mathcal{O}(1)$
42:	end if	
43:	$*(pNodo).izq).padre \leftarrow *(pNodo).padre$	$\mathcal{O}(1)$
44:	end if	
45:	else	
46:	var nuevoPNodo: puntero(nodo) $\leftarrow *(pNodo).der$	$\mathcal{O}(1)$
47:	while $*(nuevoPNodo).izq \neq \text{NULL}$ do	$\mathcal{O}(\log_2 n)$
48:	nuevoPNodo $\leftarrow *(nuevoPNodo).izq$	$\mathcal{O}(1)$
49:	end while	
50:	$*(pNodo).clave \leftarrow *(nuevoPNodo).clave$	$\mathcal{O}(1)$
51:	$*(pNodo).significado \leftarrow *(nuevoPNodo).significado$	$\mathcal{O}(1)$
52:	if $*(nuevoPNodo).der \neq \text{NULL}$ then	$\mathcal{O}(1)$
53:	if $*(nuevoPNodo).padre).izq == nuevoPNodo$ then	$\mathcal{O}(k)$
54:	$*(nuevoPNodo).padre).izq \leftarrow *(nuevoPNodo).der$	$\mathcal{O}(1)$
55:	else	
56:	$*(nuevoPNodo).padre).der \leftarrow *(nuevoPNodo).der$	$\mathcal{O}(1)$
57:	end if	
58:	$*(nuevoPNodo).der).padre \leftarrow *(nuevoPNodo).padre$	$\mathcal{O}(1)$
59:	else	
60:	if $*(nuevoPNodo).padre).izq == nuevoPNodo$ then	$\mathcal{O}(k)$
61:	$*(nuevoPNodo).padre).izq \leftarrow \text{NULL}$	$\mathcal{O}(1)$
62:	else	
63:	$*(nuevoPNodo).padre).der \leftarrow \text{NULL}$	$\mathcal{O}(1)$
64:	end if	
65:	end if	
66:	end if	
67:	end if	
68:	end if	
69:	d.tam \leftarrow d.tam - 1	$\mathcal{O}(1)$
70:	pNodo $\leftarrow *(nuevoPNodo).padre$	$\mathcal{O}(1)$
71:	while pNodo $\neq \text{NULL}$ do	$\mathcal{O}(\log_2 n * k)$
72:	var padrePNodo $\leftarrow *(pNodo).padre$	$\mathcal{O}(1)$
73:	if FACTORDESBALANCE(pNodo) > 1 then	$\mathcal{O}(1)$
74:	ROTAR(pNodo)	$\mathcal{O}(k)$
75:	else	
76:	$*(pNodo).alt \leftarrow \text{ALTURA}(pNodo)$	$\mathcal{O}(1)$
77:	end if	
78:	pNodo \leftarrow padrePNodo	$\mathcal{O}(1)$
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 todas 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  $\mathcal{O}(1)$ 
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)) $\rightarrow res : \alpha$

```
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  $\leftarrow$  CREATIT(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) $\rightarrow res : bool$

```
1: if it.#nodosRecorridos, < it.tam - 1 then  $\mathcal{O}(1)$ 
2:   res  $\leftarrow$  true  $\mathcal{O}(1)$ 
3: else
4:   res  $\leftarrow$  false  $\mathcal{O}(1)$ 
5: end if
```

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

IACTUAL(in it: itClave) $\rightarrow res : \alpha$

```
1: res  $\leftarrow$  *(it.nodoActual).clave  $\mathcal{O}(1)$ 
```

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

IAVANZAR(**in/out** *it*: itClave)

```

1: it.#nodosRecorridos  $\leftarrow$  it.#nodosRecorridos + 1  $\mathcal{O}(1)$ 
2: var itNodosNivelActual: itClave  $\leftarrow$  CREAMNODOS(it.raiz, 1, it.nivelActual)  $\mathcal{O}(n)$ 
3: var bAvanzar: bool  $\leftarrow$  true  $\mathcal{O}(1)$ 
4: while bAvanzar do  $\mathcal{O}(n)$ 
5:   AVANZAR(itNodosNivelActual)  $\mathcal{O}(1)$ 
6:   if ANTERIOR(itNodosNivelActual) == ACTUAL(it) then  $\mathcal{O}(1)$ 
7:     bAvanzar  $\leftarrow$  false  $\mathcal{O}(1)$ 
8:   else
9:     end if
10: end while
11: if HAYSIGUIENTE?(itNodosNivelActual) then  $\mathcal{O}(1)$ 
12:   it.nodoActual  $\leftarrow$  SIGUIENTE(itNodosNivelActual)  $\mathcal{O}(1)$ 
13: else
14:   it.nivelActual  $\leftarrow$  it.nivelActual + 1  $\mathcal{O}(1)$ 
15:   it.nodoActual  $\leftarrow$  SIGUIENTE(CREAMNODOS(it.raiz, 1, it.nivelActual)))  $\mathcal{O}(n)$ 
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 pertenece 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) \rightarrow *res* : Conj(nodo)

```

1: res  $\leftarrow$  VACÍO()  $\mathcal{O}(1)$ 
2: if p == NULL then  $\mathcal{O}(1)$ 
3: else
4:   if actual == destino then  $\mathcal{O}(1)$ 
5:     AGREGARATRÁS(res, p)  $\mathcal{O}(1)$ 
6:   else
7:     UNION(DAMENODOS(*(p).izq, actual + 1, destino), DAMENODOS(*(p).der, actual + 1, destino))  $\mathcal{O}(n)$ 
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 $\theta(n)$ ($f(n) = \theta(n)$), 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 $\theta(n) \in \theta(n^1)$, entonces la complejidad es $\theta(n \log_2 n)$.

IROTAR(**in/out** *p*: puntero(nodo))

```

1: if FACTORDESBALANCE(p) < -1 then  $\mathcal{O}(1)$ 
2:   if FACTORDESBALANCE(*(p).der) > 0 then  $\mathcal{O}(1)$ 
3:     res  $\leftarrow$  ROTARDOBLEIZQ(p)  $\mathcal{O}(k)$ 
4:   else
5:     res  $\leftarrow$  ROTARSIMPLEIZQ(p)  $\mathcal{O}(k)$ 
6:   end if
7: else
8:   if FACTORDESBALANCE(*(p).izq) < 0 then  $\mathcal{O}(1)$ 

```

9: $res \leftarrow \text{ROTARDOBLEDER}(p)$	$\mathcal{O}(k)$
10: else	
11: $res \leftarrow \text{ROTARSIMPLEDER}(p)$	$\mathcal{O}(k)$
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) \leftarrow p	$\mathcal{O}(1)$
2: var r2: puntero(nodo) \leftarrow *(r).der	$\mathcal{O}(1)$
3: var i: puntero(nodo) \leftarrow *(r).izq	$\mathcal{O}(1)$
4: var i2: puntero(nodo) \leftarrow *(r2).izq	$\mathcal{O}(1)$
5: var d2: puntero(nodo) \leftarrow *(r2).der	$\mathcal{O}(1)$
6: var padre: puntero(nodo) \leftarrow *(r).padre	$\mathcal{O}(1)$
7: if padre \neq NULL then	$\mathcal{O}(1)$
8: if r == *(padre).izq then	$\mathcal{O}(k)$
9: *(padre).izq \leftarrow r2	$\mathcal{O}(1)$
10: else	
11: *(padre).der \leftarrow r2	$\mathcal{O}(1)$
12: end if	
13: else	
14: end if	
15: *(r2).padre \leftarrow padre	$\mathcal{O}(1)$
16: *(r2).izq \leftarrow r	$\mathcal{O}(1)$
17: *(r).padre \leftarrow r2	$\mathcal{O}(1)$
18: *(r).der \leftarrow i2	$\mathcal{O}(1)$
19: if i2 \neq NULL then	$\mathcal{O}(1)$
20: *(i2).padre \leftarrow r	$\mathcal{O}(1)$
21: else	
22: end if	
23: *(r).alt \leftarrow ALTURA(r)	$\mathcal{O}(1)$
24: *(r2).alt \leftarrow ALTURA(r2)	$\mathcal{O}(1)$

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

Siendo k el costo de comparación de α

IROTARSIMPLEDER(in/out p: puntero(nodo))

1: var r: puntero(nodo) \leftarrow p	$\mathcal{O}(1)$
2: var r2: puntero(nodo) \leftarrow *(r).izq	$\mathcal{O}(1)$
3: var d: puntero(nodo) \leftarrow *(r).der	$\mathcal{O}(1)$
4: var i2: puntero(nodo) \leftarrow *(r2).izq	$\mathcal{O}(1)$
5: var d2: puntero(nodo) \leftarrow *(r2).der	$\mathcal{O}(1)$
6: var padre: puntero(nodo) \leftarrow *(r).padre	$\mathcal{O}(1)$
7: if padre \neq NULL then	$\mathcal{O}(1)$
8: if r == *(padre).izq then	$\mathcal{O}(k)$
9: *(padre).izq \leftarrow r2	$\mathcal{O}(1)$
10: else	
11: *(padre).der \leftarrow r2	$\mathcal{O}(1)$
12: end if	
13: else	
14: end if	
15: *(r2).padre \leftarrow padre	$\mathcal{O}(1)$
16: *(r2).der \leftarrow r	$\mathcal{O}(1)$

17: $*(r).padre \leftarrow r2$	$\mathcal{O}(1)$
18: $*(r).izq \leftarrow d2$	$\mathcal{O}(1)$
19: if $d2 \neq \text{NULL}$ then	$\mathcal{O}(1)$
20: $*(d2).padre \leftarrow r$	$\mathcal{O}(1)$
21: else	
22: end if	
23: $*(r).alt \leftarrow \text{ALTURA}(r)$	$\mathcal{O}(1)$
24: $*(r2).alt \leftarrow \text{ALTURA}(r2)$	$\mathcal{O}(1)$

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

Siendo k el costo de comparación de α

IROTARDOBLEIZQ(in/out p : puntero(nodo))

1: ROTARSIMPLEDER ($*(p).der$)	$\mathcal{O}(k)$
2: ROTARSIMPLEIZQ (p)	$\mathcal{O}(k)$

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

Siendo k el costo de comparación de α

IROTARDOBLEDER(in/out p : puntero(nodo))

1: ROTARSIMPLEIZQ ($*(p).izq$)	$\mathcal{O}(k)$
2: ROTARSIMPLEDER (p)	$\mathcal{O}(k)$

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

Siendo k el costo de comparación de α

IALTURA(in p : puntero(nodo)) $\rightarrow res$: nat

1: if $*(p).izq == \text{NULL} \wedge *(p).der == \text{NULL}$ then	$\mathcal{O}(1)$
2: $res \leftarrow 1$	$\mathcal{O}(1)$
3: else	
4: if $*(p).izq \neq \text{NULL} \wedge *(p).der == \text{NULL}$ then	$\mathcal{O}(1)$
5: $res \leftarrow *(p).izq.alt + 1$	$\mathcal{O}(1)$
6: else	
7: if $*(p).izq == \text{NULL} \wedge *(p).der \neq \text{NULL}$ then	$\mathcal{O}(1)$
8: $res \leftarrow *(p).der.alt + 1$	$\mathcal{O}(1)$
9: else	
10: $res \leftarrow \max(*(p).izq.alt, *(p).der.alt) + 1$	$\mathcal{O}(1)$
11: end if	
12: end if	
13: end if	

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

IFACTORDESBALANCE(in p : puntero(nodo)) $\rightarrow res$: nat

1: if $*(p).izq == \text{NULL} \wedge *(p).der == \text{NULL}$ then	$\mathcal{O}(1)$
2: $res \leftarrow 0$	$\mathcal{O}(1)$
3: else	
4: if $*(p).izq \neq \text{NULL} \wedge *(p).der == \text{NULL}$ then	$\mathcal{O}(1)$
5: $res \leftarrow *(p).izq.alt$	$\mathcal{O}(1)$
6: else	
7: if $*(p).izq == \text{NULL} \wedge *(p).der \neq \text{NULL}$ then	$\mathcal{O}(1)$
8: $res \leftarrow -*(p).der.alt$	$\mathcal{O}(1)$
9: else	

10: $res \leftarrow *(*(p).izq).alt - *(*(p).der).alt$	$\mathcal{O}(1)$
11: end if	
12: end if	
13: end if	

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

ISIGUIENTES(**in** $it : itClave$) $\rightarrow res : Lista\ Enlazada$

1: var itVar: itClave $\leftarrow it$	$\mathcal{O}(1)$
2: var siguientes: Lista Enlazada $\leftarrow VACIA()$	$\mathcal{O}(1)$
3: while HAYMAS?(itVar) do	$\mathcal{O}(it.tam^2)$
4: AGREGARATRAS(siguientes, ACTUAL(itVar))	$\mathcal{O}(1)$
5: AVANZAR(it)	$\mathcal{O}(it.tam)$
6: end while	
7: $res \leftarrow siguientes$	$\mathcal{O}(1)$

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

6. Extensión de Lista Enlazada(α)

6.1. Interfaz

Interfaz

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

géneros: lista , $\text{itLista}(\alpha)$.

Operaciones básicas de lista

$\text{PERTENECE?}(\text{in } l : \text{lista}, \text{in } e : \alpha) \rightarrow res : \text{bool}$

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

Post $\equiv \{res =_{\text{obs}} \text{está?}(l, e)\}$

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

Descripción: Devuelve true o false según si el elemento pertenece o no a la lista

6.2. Algoritmos

Algoritmos

$\text{PERTENECE?}(\text{in } l : \text{lista}(\alpha), \text{in } e : \text{lista}) \rightarrow res : \text{bool}$

1: var $itLista \leftarrow \text{CrearIt}(l)$	$\mathcal{O}(1)$
2: $res \leftarrow \text{false}$	$\mathcal{O}(1)$
3: while $\text{HaySiguiente}(itLista)$ AND $\neg res$ do	$\mathcal{O}(1)$
4: if $\text{Siguiente}(itLista) == e$ then	$\mathcal{O}(1)$
5: $res \leftarrow \text{true}$	$\mathcal{O}(1)$
6: end if	
7: $\text{Avanzar}(itLista)$	$\mathcal{O}(1)$
8: end while	

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

7. Extensión de Conjunto Lineal(α)

7.1. Interfaz

Interfaz

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

géneros: conj , $\text{itConj}(\alpha)$.

Operaciones básicas de Conjunto

$\text{UNION}(\text{in/out } c : \text{conj}(\alpha), \text{in } d : \text{conj}(\alpha)) \rightarrow res : \text{itConj}(\alpha)$

Pre $\equiv \{c =_{\text{obs}} c_0\}$

Post $\equiv \{res =_{\text{obs}} \text{crearItBi}(c \cup d)\}$

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

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

Pre $\equiv \{\#(c) > 0\}$

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

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

Descripción: Devuelve un elemento cualquiera del conjunto

7.2. Algoritmos

Algoritmos

$\text{UNION}(\text{in/out } c : \text{conj}(\alpha), \text{in } d : \text{conj}(\alpha)) \rightarrow res : \text{itConj}(\alpha)$

1: var $itConj \leftarrow \text{CrearIt}(d)$	$\mathcal{O}(1)$
2: while $\text{HaySiguiente}(itConj)$ do	$\mathcal{O}(1)$
3: Agregar(c , $\text{Siguiente}(itConj)$)	$\mathcal{O}(1)$
4: Avanzar($itConj$)	$\mathcal{O}(1)$
5: end while	
6: var $res \leftarrow \text{CrearIt}(c)$	$\mathcal{O}(1)$

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

$\text{DAMEUNO}(\text{in } c : \text{conj}(\alpha))$

1: var $itConj \leftarrow \text{CrearIt}(c)$	$\mathcal{O}(1)$
2: $res \leftarrow \text{Siguiente}(itConj)$	$\mathcal{O}(1)$

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