

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. Auxiliares	6
2.2. Representacion	6
2.3. InvRep y Abs	6
2.4. Algoritmos	7
3. DCNet	10
3.1. Interfaz	10
3.2. Representacion	11
3.3. InvRep y Abs	11
3.4. Algoritmos	13
4. Diccionario String(α)	16
5. Diccionario Rápido	17
5.1. Interfaz	17
5.2. Auxiliares	18
5.3. Representación	19
5.4. InvRep y Abs	19
5.5. Algoritmos	20

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**
 $\text{long}(\text{restricciones}(m)) - 1$
else
 $\text{nroConexion}(e_1, e_2, m) - 1$
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 $d \longrightarrow \text{Clave}$ $\{\neg \text{vacío}(d)\}$

axiomas

vacío?(vacío) $\equiv \text{true}$

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

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

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)   $\equiv$   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

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

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

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) $\rightarrow res$: itConj(compu)

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

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

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) $\rightarrow res$: conj(lista(compu))

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

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

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

2.1. Auxiliares

Operaciones auxiliares

CAMINOSMINIMOS(in r : red, in c_1 : compu, in c_2 : compu) $\rightarrow res$: conj(lista)
Pre $\equiv \{\{c_1, c_2\} \subseteq computadoras(r)\}$
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

2.2. Representacion

Representación

red se representa con e_red

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

2.3. InvRep y Abs

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

Rep : e_mapa \rightarrow bool
Rep(m) $\equiv \text{true} \iff$
 $m.\text{estaciones} = \text{claves}(m.\text{uniones}) \wedge$ 1.
 $m.\#sendas = \#sendasPorDos(m.\text{estaciones}, m.\text{uniones}) / 2 \wedge m.\#sendas \leq \text{long}(m.\text{sendas})$ 2. 5.
 \wedge_L
 $(\forall e1, e2: \text{string})(e1 \in \text{claves}(m.\text{uniones}) \wedge_L e2 \in \text{claves}(\text{obtener}(e1, m.\text{uniones})) \Rightarrow_L$
 $e2 \in \text{claves}(m.\text{uniones}) \wedge_L e1 \in \text{claves}(\text{obtener}(e2, m.\text{uniones})) \wedge_L$
 $\text{obtener}(e2, \text{obtener}(e1, m.\text{uniones})) = \text{obtener}(e1, \text{obtener}(e2, m.\text{uniones})) \wedge$ 3. 4.
 $\text{obtener}(e2, \text{obtener}(e1, m.\text{uniones})) < m.\#sendas \wedge$
 $(\forall e1, e2, e3, e4: \text{string})(e1 \in \text{claves}(m.\text{uniones}) \wedge_L e2 \in \text{claves}(\text{obtener}(e1, m.\text{uniones})) \wedge$
 $e3 \in \text{claves}(m.\text{uniones}) \wedge_L e4 \in \text{claves}(\text{obtener}(e3, m.\text{uniones}))) \Rightarrow_L$
 $(\text{obtener}(e2, \text{obtener}(e1, m.\text{uniones})) = \text{obtener}(e4, \text{obtener}(e3, m.\text{uniones}))) \iff$
 $(e1 = e3 \wedge e2 = e4) \vee (e1 = e4 \wedge e2 = e3))$ 3.

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

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

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

2.4. Algoritmos

Algoritmos

$I\text{COMPUTADORAS}(\text{in } r : \text{red}) \rightarrow res : \text{itConj}(\text{Compu})$

1: $res \leftarrow \text{CrearIt}(r.computadoras)$

$\mathcal{O}(1)$

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

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

1: $res \leftarrow \text{Definido?}(\text{Significado}(r.vecinosEInterfaces, c_1.ip).interfaces, c_2.ip)$

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

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

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

1: $res \leftarrow \text{Significado}(\text{Significado}(r.vecinosEInterfaces, c_1.ip).interfaces, c_2.ip)$

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

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

$I\text{INICIARRED}() \rightarrow res : \text{red}$

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

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

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

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

$I\text{AGREGARCOMPUTADORA}(\text{in/out } r : \text{red}, \text{in } c : \text{compu})$

1: $\text{Agregar}(r.computadoras, c)$

$\mathcal{O}(1)$

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

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

3: $\text{Definir}(r.deOrigenADestino, c.ip, \text{Vacio}())$

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

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

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

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

```

ICONECTAR(in/out r: red, in c1: compu, in i1: interfaz, in c2: compu, in i2: interfaz)
1: var tupSig1:tupla ← Significado(r.vecinosEInterfases, c1.ip)
2: Definir(tupSig1.interfases, c2.ip, i1) O(|c1| + |c2| + 1)
3: Agregar(tupSig1.compusVecinas, c2) O(1)
4: var tupSig2:tupla ← Significado(r.vecinosEInterfases, c2.ip)
5: Definir(tupSig2.interfases, c1.ip, i2) O(|c1| + |c2| + 1)
6: Agregar(tupSig2.compusVecinas, c1) O(1)
7: Definir(Significado(r.deOrigenADestino, c1.ip), c2.ip, ArmarCaminosMinimos(r, c1, c2)) O(1)
8: Definir(Significado(r.deOrigenADestino, c2.ip), c1.ip, ArmarCaminosMinimos(r, c2, c1)) O(1)

```

Complejidad: $O(|e_1| + |e_2|)$

$O(|e_1| + |e_2|) + O(|e_1| + |e_2|) + O(1) + O(1) =$
 $2 * O(1) + 2 * O(|e_1| + |e_2|) =$
 $2 * O(|e_1| + |e_2|) = O(|e_1| + |e_2|)$

```

IVECINOS(in r: red, in c: compu) → res: itConj(compu)
1: res ← crearIt((Significado(r.vecinosEInterfases, c.ip)).compusVecinas)

```

Complejidad:

```

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

```

Complejidad:

```

ICAMINOSMINIMOS(in r: red, in c1: compu, in c2: compu) → res: conj(lista(compu))
1: res ← Significado(Significado(r.deOrigenADestino, c1.ip), c2.ip) O(|c1| + |c2|)

```

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:

```

IARMARCAMINOS(in r: red, in c1: compu, in c2: compu) → res: conj(lista)
1: var parcial:lista ← Vacía() O(1)
2: AgregarAtras(parcial, c1) O(1)
3: res ← TODOSLOS CAMINOS(r, c1, c2, parcial) O(1)

```


Complejidad:

```
ITodosLosCaminos(in  $r$ : red, in  $c_1$ : compu, in  $c_2$ : compu, in  $pacial$ : lista(compu))  $\rightarrow res$  : conj(lista)
1:  $res \leftarrow \text{Vacio}()$   $\mathcal{O}(1)$ 
2: if Pertenece?(Vecinos( $r$ ,  $c_1$ ),  $c_2$ ) then  $\mathcal{O}(1)$ 
3:   AgregarAtras( $pacial$ ,  $c_2$ )  $\mathcal{O}(1)$ 
4:   Agregar( $res$ ,  $pacial$ )  $\mathcal{O}(1)$ 
5: else
6:   var  $itVecinos$ :itConj  $\leftarrow$  CrearIt(Vecinos( $r$ ,  $c_1$ ))  $\mathcal{O}(1)$ 
7:   while HaySiguiente?( $itVecinos$ ) do  $\mathcal{O}(1)$ 
8:     if  $\neg$ Pertenece?( $pacial$ , Siguiente( $itVecinos$ )) then  $\mathcal{O}(1)$ 
9:       var  $auxParcial$ :lista  $\leftarrow$   $pacial$   $\mathcal{O}(1)$ 
10:      AgregarAtras( $auxParcial$ , Siguiente( $itVecinos$ ))  $\mathcal{O}(1)$ 
11:      Unir( $res$ , TodosLosCaminos( $r$ , Siguiente( $itVecinos$ ),  $c_2$ ,  $auxParcial$ ))  $\mathcal{O}(1)$ 
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, ITERADOR UNIDIRECCIONAL(α).

géneros: dcnet.

Operaciones básicas de DCNet

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

Pre $\equiv \{true\}$

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

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

Descripción: Devuelve la red del dcnet.

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

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

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

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 $\equiv \{c \in \text{computadoras}(\text{red}(d))\}$

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

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 $\equiv \{c \in \text{computadoras}(\text{red}(d))\}$

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

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

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

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

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

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

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

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 $\equiv \{d =_{\text{obs}} \text{avanzarSegundo}(c_0)\}$

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

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

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

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

dcnet se representa con e_{dc}

donde e_{dc} es $\text{tupla}(\text{red: red},$
 $\text{MasEnviante: tupla}(\text{compu: compu}, \text{enviados: nat}),$
 $\text{CompYPaq: DiccString}(\text{compu: compu}, \text{tupla}(\text{MasPriori: DiccRapido}(\text{prioridad}$
 $:\text{nat}, \text{PaqdePriori: conj}(\text{paquete})), \text{PaqYCam: DiccRapido}(\text{paq: paquete}, \text{CamRecorri-$
 $\text{do: secu}(\text{compu})$
 $)$

3.3. InvRep y Abs

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

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

$\text{estaEnColaP?} : \text{ColaPri} \times \text{Nat} \longrightarrow \text{Bool}$

```

    estaEnColaP?(cp, n) ≡
if vacia?(cp) then
    false
else
    if desencolar(cp) = n then
        true
    else
        estaEnColaP?(Eliminar(cp, desencolar(cp)), n)
    fi
fi

```

```

Abs          : e_cr c          → ciudad                                {Rep(c)}
Abs(c) =obs u: ciudad |
    c.#RURHistoricos = ProximoRUR(U) ∧ c.mapa = mapa(u) ∧L
    robots(u) = RURQueEstan(c.RURs) ∧L
    (∀ n:Nat) (n ∈ robots(u) ⇒L estacion(n,u) = c.RURs[n].e ∧
    tags(n,u) = c.RURs[n].carac ∧ #infracciones(n,u) = c.RURs[n].inf)

```

```

RURQueEstan: secu(tupla)      → Conj(RUR)

```

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

```

RURQueEstan(s) ≡ if vacia?(s) then
    ∅
else
    if Π2(prim(fin(s))) then
        Π1(prim(fin(s))) ∪ RURQueEstan(fin(s))
    else
        RURQueEstan(fin(s))
    fi
fi

```

it se representa con e_it

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

```

Rep          : e_it          → bool
Rep(it)      ≡ true ⇔ it.i ≤ it.maxI ∧ maxI = ciudad.#RURHistoricos
Abs          : e_it u        → itUni(α)                                {Rep(u)}
Abs(u) =obs it: itUni(α) | (HayMas?(u) ∧L Actual(u) = ciudad.RURs[it.i] ∧ Siguientes(u, ∅) = VSiguientes(ciudad,
    it.i++, ∅) ∨ (¬HayMas?(u))

```

```

Siguientes   : itUniu        × → conj(RURs)
               conj(RURs)cr

```

```

Siguientes(u, cr) ≡ if HayMas(u)? then
    Ag(Actual(Avanzar(u)), Siguientes(Avanzar(u), cr))
else
    Ag(∅, cr)
fi

```

```

VSiguientes  : ciudadc × Nati × → conj(RURs)
               conj(RURs)cr

```

```

VSiguientes(u, i, cr) ≡
if i < c.#RURHistoricos then Ag(c.RURs[i], VSiguientes(u, i++, cr)) else Ag(∅, cr) fi

```

3.4. Algoritmos

Algoritmos

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

1: $res \leftarrow (d.red)$

$\mathcal{O}(1)$

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

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

1: var $it \leftarrow \text{COMPUTADORAS}(d.red)$

$\mathcal{O}(1)$

2: var $esta$: **bool** $\leftarrow \text{false}$

3: **while** **HAYSIGUIENTE**(it) $\wedge \neg esta$ **do**

$\mathcal{O}(n)$

4: var: **diccRapido** $diccpaq \leftarrow \text{OBTENER}(\text{SIGUIENTE}(it).id, d.\text{CompYPaq}).\text{PaqYCam}$

$\mathcal{O}(L)$

5: **if** **DEF?**($p, diccpaq$) **then**

$\mathcal{O}(L + \log_2(N))$

6: $esta \leftarrow \text{true}$

$\mathcal{O}(1)$

7: $res \leftarrow \text{OBTENER}(p, diccpaq).\text{CamRecorrido}$

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

8: **end if**

9: **AVANZAR**(it)

$\mathcal{O}(1)$

10: **end while**

Complejidad: $\mathcal{O}()$

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

1: $res \leftarrow \text{OBTENER}(c.id, d.\text{CompYPaq}).\text{Enviados}$

$\mathcal{O}(L)$

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

Siendo L la longitud de el ID de c

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

1: $res \leftarrow \text{CLAVES}(\text{OBTENER}(c.id, d.\text{CompYPaq}).\text{PaqYCam})$

$\mathcal{O}(L)$

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

Siendo L la longitud del ID de c

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

1: $d.red \leftarrow r$

$\mathcal{O}(NOSE)$

2: var $it \leftarrow \text{COMPUTADORAS}(red)$

$\mathcal{O}(1)$

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**($\text{SIGUIENTE}(it).id, \text{tupla}(\text{VACIO}(), \text{VACIO}(), 0), d.\text{CompyPaq}$)

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

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 ID mas largo de ellas.

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  $\neg \text{DEF?}(p.\text{prioridad}, \text{diccprio})$  then  $\mathcal{O}(\log_2(s))$ 
4:   DEFINIR( $p$ .prioridad, AGREGAR(VACIO(),  $p$ ), diccprio)  $\mathcal{O}(\log_2(s))$ 
5: else
6:   DEFINIR( $p$ .prioridad, AGREGAR(OBTENER( $p$ .prioridad, diccprio),  $p$ ), diccprio)  $\mathcal{O}(\log_2(s))$ 
7: end if
8: DEFINIR( $p$ , dicccam, AGREGARATRAS(<>,  $p$ .origen))  $\mathcal{O}(\log_2(k))$ 

```

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

IAVANZARSEGUNDO(in/out d : dcnet)

```

1: var it ← COMPUTADORAS(red)  $\mathcal{O}(1)$ 
2: var aux ← VACIA()  $\mathcal{O}(1)$ 
3: while HAYSIGUIENTE(it) do  $\mathcal{O}(N)$ 
4:   var diccprio: diccRapido ← OBTENER(SIGUIENTE(it).id,  $d$ .CompYPaq).MasPriori  $\mathcal{O}(L)$ 
5:   var dicccam: diccRapido ← OBTENER(SIGUIENTE(it).id,  $d$ .CompYPaq).PaqYCam  $\mathcal{O}(L)$ 
6:   if  $\neg \text{VACIO?}(\text{diccprio})$  then  $\mathcal{O}(1)$ 
7:     var paq: paquete ← PRIMERO(OBTENER(DAMEMAX(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(DAMEMAX(diccprio), diccprio), paq)  $\mathcal{O}(\log_2(k) + \log_2(k) + 1)$ 
10:    if ESVACIO?(OBTENER(DAMEMAX(diccprio), diccprio) then  $\mathcal{O}(\log_2(k))$ 
11:      BORRAR(DAMEMAX(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 ← 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 ← CREAMIT(aux)  $\mathcal{O}(1)$ 
22: while HAYSIGUIENTE(itaux) do  $\mathcal{O}(Nk)$ 
23:   var proxpc: compu ← PRIMERO(SIGUIENTE(CAMINOSMINIMOS( $d$ .red, itaux.pcant, itaux.destino)))  $\mathcal{O}(L_1 + L_2)$ 
24:   var diccprio: diccRapido ← OBTENER(proxpc.id,  $d$ .CompYPaq).MasPriori  $\mathcal{O}(L)$ 
25:   var dicccam: diccRapido ← OBTENER(proxpc.id,  $d$ .CompYPaq).PaqYCam  $\mathcal{O}(L)$ 
26:   if DEF?((itaux.paq).prioridad, diccprio) then  $\mathcal{O}(\log_2(k))$ 
27:     var mismaprio: conj(paquetes) ← AGREGAR(OBTENER(it3.paq.prioridad, diccprio), it3.paq)  $\mathcal{O}(\log_2(k))$ 
28:     DEFINIR((it3.paq).prioridad, mismaprio, diccprio)  $\mathcal{O}(\log_2(k))$ 
29:   else
30:     DEFINIR(it3.prioridad, AGREGAR(VACIO(), it3.paq), diccprio)  $\mathcal{O}(\log_2(k))$ 
31:   end if
32:   DEFINIR( $p$ .paq, AGREGARATRAS(it3.camrecorrido, proxpc), dicccam)  $\mathcal{O}(\log_2(k))$ 
33:   ELIMINARSIGUIENTE(it3)  $\mathcal{O}(1)$ 
34:   AVANZAR(it3)  $\mathcal{O}(1)$ 
35: end while

```

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

IPAQUETEENTRANSITO? (in d : dcnet , in p : paquete) $\rightarrow res$: bool	
1: var $it \leftarrow$ CREAMIT(COMPUTADORAS($d.red$))	$\mathcal{O}(1)$
2: var $esta$: bool \leftarrow false	$\mathcal{O}(1)$
3: while HAYSIGUIENTE(it) $\wedge \neg esta$ do	$\mathcal{O}()$
4: $esta \leftarrow$ DEF?(OBTENER($d.CompYPaq,i.id$).PaqYCam , p)	$\mathcal{O}(\log_2(k))$
5: AVANZAR(it)	$\mathcal{O}(1)$
6: end while	
7: $res \leftarrow esta$	$\mathcal{O}(1)$
 Complejidad: $\mathcal{O}(N * \log(k))$	

ILAQUEMASENVIO (in d : dcnet) $\rightarrow res$: compu	
1: $res \leftarrow (d.MasEnviante).compu$	$\mathcal{O}(1)$
 Complejidad: $\mathcal{O}(1)$	

4. Diccionario $\text{String}(\alpha)$

Interfaz

se explica con: $\text{DICCIONARIO}(\text{STRING}, \alpha)$.

géneros: $\text{diccString}(\alpha)$.

Operaciones básicas de Diccionario $\text{String}(\alpha)$

$\text{DEF?}(\text{in } clv : \text{string}, \text{in } d : \text{diccString}(\alpha)) \rightarrow res : \text{bool}$

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

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

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

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

$\text{OBTENER}(\text{in } clv : \text{string}, \text{in } d : \text{diccString}(\alpha)) \rightarrow res : \alpha$

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

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

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

Descripción: Devuelve el significado de la clave.

$\text{VACÍO}() \rightarrow res : \text{diccString}(\alpha)$

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

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

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

Descripción: Crea nuevo diccionario vacío.

$\text{DEFINIR}(\text{in } clv : \text{string}, \text{in } def : \alpha, \text{in/out } d : \text{diccString}(\alpha))$

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

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

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

Descripción: Agrega una nueva definición.

$\text{BORRAR}(\text{in } clv : \text{string}, \text{in/out } d : \text{diccString}(\alpha))$

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

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

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

Descripción: Devuelve la definición correspondiente a la clave.

5. Diccionario Rápido

5.1. Interfaz

Interfaz

se explica con: `DICCIONARIO(CLAVE, SIGNIFICADO)`.

géneros: `diccRapido(α, β)`.

Operaciones básicas de Diccionario Rápido(α, β)

DEF?(**in** $c : \alpha$, **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

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

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 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

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 : diccRapido(α, β)) $\rightarrow res$: 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

HAYMAS?(**in** it : itClave) $\rightarrow res$: 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 : 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

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

DAMENODOS(**in** p : puntero(nodo), **in** $actual$: nat, **in** $destino$: nat) $\rightarrow res$: conj(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 : puntero(nodo))

Pre $\equiv \{p =_{\text{obs}} p_0 \wedge p \neq \text{NULL} \wedge_L ($

$*(p).der \neq \text{NULL} \vee$

$*(p).izq \neq \text{NULL} \vee$

$*(p).der \neq \text{NULL} \wedge_L (*(p).der).izq \neq \text{NULL}) \vee$

$*(p).izq \neq \text{NULL} \wedge_L (*(p).izq).der \neq \text{NULL})\}$

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

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

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)) $\rightarrow res : \text{nat}$

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)) $\rightarrow res : \text{nat}$

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

5.3. Representación

Representación

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

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

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

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

5.4. InvRep y Abs

InvRep en lenguaje coloquial:

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

12. La componente "alt" de cada nodo es igual a la cantidad de nodos que hay que "bajar" para llegar a su hoja mas lejana + 1. Vale aclarar que el nodo hoja tiene la componente "alt" igual a 1.
13. Para todo nodo, la diferencia, en módulo, de la altura entre sus subárboles es menor o igual a 1.

Abs:

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

5.5. Algoritmos

Algoritmos

```

IDEF?(in c:  $\alpha$ , in d: diccRapido( $\alpha, \beta$ ))  $\rightarrow res : \text{bool}$ 
1: var pNodo: puntero(nodo)  $\leftarrow$  d.raiz  $\mathcal{O}(1)$ 
2: while *(pNodo).clave != NULL do  $\mathcal{O}(\log_2 n)$ 
3:   if *(pNodo).clave == c then  $\mathcal{O}(1)$ 
4:     res  $\leftarrow$  true  $\mathcal{O}(1)$ 
5:     return res  $\mathcal{O}(1)$ 
6:   else
7:     if c > *(pNodo).clave then  $\mathcal{O}(1)$ 
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: $\mathcal{O}(\log_2 n)$

Siendo n la cantidad de nodos.

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

```

IOBTENER(in c:  $\alpha$ , 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  $\mathcal{O}(\log_2 n)$ 
3:   if c > *(pNodo).clave then  $\mathcal{O}(1)$ 
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: $\mathcal{O}(\log_2 n)$

Siendo n la cantidad de nodos.

Es un algoritmo muy parecido al de DEF?. Nuevamente ignoraremos los condicionales y las asignaciones, dado

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

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

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

$\mathcal{O}(1)$

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  $\leftarrow \&\text{tupla}(c, s, \text{NULL}, \text{NULL}, \text{NULL}, 1)$   $\mathcal{O}(1)$ 
4:   d.tam  $\leftarrow 1$   $\mathcal{O}(1)$ 
5: else
6:   if DEF?(c, d) then  $\mathcal{O}(\log_2 n)$ 
7:     var pNodo: puntero(nodo)  $\leftarrow d.raiz$   $\mathcal{O}(1)$ 
8:     while *(pNodo).clave != c do  $\mathcal{O}(\log_2 n)$ 
9:       if c > *(pNodo).clave then  $\mathcal{O}(1)$ 
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  $\leftarrow \text{true}$   $\mathcal{O}(1)$ 
18:    var pNodo: puntero(nodo)  $\leftarrow d.raiz$   $\mathcal{O}(1)$ 
19:    var camino: arreglo[ $\lceil \log_2 (d.tam) \rceil + 1$ ] de puntero(nodo)  $\mathcal{O}(\log_2 n)$ 
20:    var nroCamino: nat  $\leftarrow 0$   $\mathcal{O}(1)$ 
21:    camino[nroCamino]  $\leftarrow pNodo$   $\mathcal{O}(1)$ 
22:    while seguir == true do  $\mathcal{O}(\log_2 n)$ 
23:      if c > *(pNodo).clave  $\wedge$  *(pNodo).der == NULL then  $\mathcal{O}(1)$ 
24:        if *(pNodo).izq == NULL then  $\mathcal{O}(1)$ 
25:          *(pNodo).alt  $\leftarrow 2$   $\mathcal{O}(1)$ 
26:        else
27:          end if
28:        *(pNodo).der  $\leftarrow \&\text{tupla}(c, s, pNodo, \text{NULL}, \text{NULL}, 1)$   $\mathcal{O}(1)$ 
29:        nroCamino  $\leftarrow \text{nroCamino} + 1$   $\mathcal{O}(1)$ 
30:        camino[nroCamino]  $\leftarrow *(pNodo).der$   $\mathcal{O}(1)$ 
31:        seguir  $\leftarrow \text{false}$   $\mathcal{O}(1)$ 
32:      else
33:        if c > *(pNodo).clave  $\wedge$  *(pNodo).der != NULL then  $\mathcal{O}(1)$ 
34:          if *(pNodo).izq == NULL then  $\mathcal{O}(1)$ 
35:            *(pNodo).alt  $\leftarrow *(pNodo).alt + 1$   $\mathcal{O}(1)$ 
36:          else
37:            *(pNodo).alt  $\leftarrow \max(*(pNodo).izq).alt, *(pNodo).der).alt + 1$   $\mathcal{O}(1)$ 
38:          end if
39:          pNodo  $\leftarrow *(pNodo).der$   $\mathcal{O}(1)$ 
40:          nroCamino  $\leftarrow \text{nroCamino} + 1$   $\mathcal{O}(1)$ 
41:          camino[nroCamino]  $\leftarrow pNodo$   $\mathcal{O}(1)$ 
42:        else
43:          if c < *(pNodo).clave  $\wedge$  *(pNodo).izq == NULL then  $\mathcal{O}(1)$ 

```

44:	if *(pNodo).der == NULL then	$\mathcal{O}(1)$
45:	*(pNodo).alt \leftarrow 2	$\mathcal{O}(1)$
46:	else	
47:	end if	
48:	*(pNodo).izq \leftarrow &tupla(c, s, pNodo, NULL, NULL, 1)	$\mathcal{O}(1)$
49:	nroCamino \leftarrow nroCamino + 1	$\mathcal{O}(1)$
50:	camino[nroCamino] \leftarrow *(pNodo).izq	$\mathcal{O}(1)$
51:	seguir \leftarrow false	$\mathcal{O}(1)$
52:	else	
53:	if *(pNodo).der == NULL then	$\mathcal{O}(1)$
54:	*(pNodo).alt \leftarrow *(pNodo).alt + 1	$\mathcal{O}(1)$
55:	else	
56:	*(pNodo).alt \leftarrow max(*(pNodo).izq).alt + 1, *(pNodo).der).alt)	$\mathcal{O}(1)$
57:	end if	
58:	pNodo \leftarrow *(pNodo).izq	$\mathcal{O}(1)$
59:	nroCamino \leftarrow nroCamino + 1	$\mathcal{O}(1)$
60:	camino[nroCamino] \leftarrow pNodo	$\mathcal{O}(1)$
61:	end if	
62:	end if	
63:	end if	
64:	end while	
65:	d.tam \leftarrow d.tam + 1	$\mathcal{O}(1)$
66:	while nroCamino \geq 0 do	$\mathcal{O}(\log_2 n)$
67:	pNodo \leftarrow camino[nroCamino]	$\mathcal{O}(1)$
68:	if FACTORDESBALANCE(pNodo) > 1 then	$\mathcal{O}(1)$
69:	ROTAR(pNodo)	$\mathcal{O}(1)$
70:	else	
71:	end if	
72:	nroCamino \leftarrow nroCamino - 1	$\mathcal{O}(1)$
73:	end while	
74:	end if	
75:	end if	

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

Siendo n la cantidad de nodos.

En este algoritmo, tomaremos en cuenta las complejidades de tres casos e ignoraremos los condicionales y asignaciones (dado que son constantes).

-El primer caso es cuando se quiera definir en un diccionario vacío, esto es $\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) = 3 * \mathcal{O}(1) = \mathcal{O}(1)$

-El segundo caso es cuando se quiera definir una clave que ya estaba definida previamente, aquí ignoraremos las asignaciones y condicionales (cuyas complejidades son $\mathcal{O}(1)$), y nos centraremos en el uso de DEF? y el ciclo. DEF? sabemos que toma tiempo logarítmico, y en cuanto al ciclo, sabemos que tomará tiempo logarítmico también, porque iterará hasta buscar el nodo buscado. Esto es: $\mathcal{O}(\log_2 n) + \mathcal{O}(\log_2 n) = 2 * \mathcal{O}(\log_2 n) = \mathcal{O}(\log_2 n)$

-El tercer caso es cuando se quiera definir una clave que no estaba definida anteriormente. Nuevamente ignoraremos las asignaciones y condicionales, y nos centraremos en la creación del arreglo "camino", y los siguientes dos ciclos. Dado que se quiere crear un arreglo donde se guarden los punteros a nodos recorridos, como máximo en éste se guardarán $\log_2 n + 1$ nodos (porque en peor caso tendríamos que descender hasta la hoja más lejana para insertar). Por eso, basta con crear el arreglo con $\log_2 n + 1$ posiciones, y esto cuesta $\mathcal{O}(\log_2 n)$. Luego, el primer ciclo consiste en iterar hasta llegar a la posición donde queremos insertar el nuevo nodo, nuevamente esto es $\mathcal{O}(\log_2 n)$ porque en peor caso tendríamos que descender hasta la hoja más lejana. Por último, el último ciclo recorre el arreglo "camino" de atrás hacia adelante (en realidad no todo el arreglo, sino desde el último elemento insertado en él), y dado que éste tiene (a lo sumo) $\log_2 n$ elementos, esto es $\mathcal{O}(\log_2 n)$. Finalmente, dado que este caso tiene éstas tres complejidades no anidadas: $\mathcal{O}(\log_2 n) + \mathcal{O}(\log_2 n) + \mathcal{O}(\log_2 n) = 3 * \mathcal{O}(\log_2 n) = \mathcal{O}(\log_2 n)$.

-Ahora, como teníamos tres casos, la complejidad es el máximo de ellos: $\max(\mathcal{O}(1), \mathcal{O}(\log_2 n), \mathcal{O}(\log_2 n)) = \mathcal{O}(\log_2 n)$

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

1: var pNodo: puntero(nodo) \leftarrow d.raiz	$\mathcal{O}(1)$
2: var camino: arreglo[$\lceil \log_2 (d.tam) \rceil + 1$] de puntero(nodo)	$\mathcal{O}(\log_2 n)$
3: var nroCamino: nat \leftarrow 0	$\mathcal{O}(1)$
4: camino[nroCamino] \leftarrow pNodo	$\mathcal{O}(1)$
5: while $c \neq *(pNodo).clave$ do	$\mathcal{O}(\log_2 n)$
6: if $c > *(pNodo).clave$ then	$\mathcal{O}(1)$
7: if $*(pNodo).izq == \text{NULL}$ then	$\mathcal{O}(1)$
8: $*(pNodo).alt \leftarrow *(pNodo).alt - 1$	$\mathcal{O}(1)$
9: else	
10: $*(pNodo).alt \leftarrow \max(*(pNodo).izq).alt, *(pNodo).der).alt - 1$	$\mathcal{O}(1)$
11: end if	
12: $pNodo \leftarrow *(pNodo).der$	$\mathcal{O}(1)$
13: $nroCamino \leftarrow nroCamino + 1$	$\mathcal{O}(1)$
14: camino[nroCamino] \leftarrow pNodo	$\mathcal{O}(1)$
15: else	
16: if $*(pNodo).der == \text{NULL}$ then	$\mathcal{O}(1)$
17: $*(pNodo).alt \leftarrow *(pNodo).alt - 1$	$\mathcal{O}(1)$
18: else	
19: $*(pNodo).alt \leftarrow \max(*(pNodo).izq).alt - 1, *(pNodo).der).alt$	$\mathcal{O}(1)$
20: end if	
21: $pNodo \leftarrow *(pNodo).izq$	$\mathcal{O}(1)$
22: $nroCamino \leftarrow nroCamino + 1$	$\mathcal{O}(1)$
23: camino[nroCamino] \leftarrow pNodo	$\mathcal{O}(1)$
24: end if	
25: end while	
26: if $*(pNodo).izq == \text{NULL} \wedge *(pNodo).der == \text{NULL}$ then	$\mathcal{O}(1)$
27: if $*(pNodo).padre == \text{NULL}$ then	$\mathcal{O}(1)$
28: $d.raiz \leftarrow \text{NULL}$	$\mathcal{O}(1)$
29: delete pNodo	$\mathcal{O}(1)$
30: else	
31: if $*(pNodo).clave == (*(pNodo).padre).izq).clave$ then	$\mathcal{O}(1)$
32: $*(pNodo).padre).izq \leftarrow \text{NULL}$	$\mathcal{O}(1)$
33: else	
34: $*(pNodo).padre).der \leftarrow \text{NULL}$	$\mathcal{O}(1)$
35: end if	
36: delete pNodo $\mathcal{O}(1)$	
37: end if	
38: else	
39: if $*(pNodo).izq == \text{NULL} \wedge *(pNodo).der \neq \text{NULL}$ then	$\mathcal{O}(1)$
40: if $*(pNodo).padre == \text{NULL}$ then	$\mathcal{O}(1)$
41: $*(pNodo).der).padre \leftarrow \text{NULL}$	$\mathcal{O}(1)$
42: $d.raiz \leftarrow *(pNodo).der$	$\mathcal{O}(1)$
43: delete pNodo	$\mathcal{O}(1)$
44: else	
45: if $*(pNodo).clave == (*(pNodo).padre).izq).clave$ then	$\mathcal{O}(1)$
46: $*(pNodo).padre).izq \leftarrow *(pNodo).der$	$\mathcal{O}(1)$
47: else	
48: $*(pNodo).padre).der \leftarrow *(pNodo).der$	$\mathcal{O}(1)$
49: end if	
50: $*(pNodo).der).padre \leftarrow *(pNodo).padre$	$\mathcal{O}(1)$
51: delete pNodo	$\mathcal{O}(1)$
52: end if	
53: else	
54: if $*(pNodo).izq \neq \text{NULL} \wedge *(pNodo).der == \text{NULL}$ then	
55: if $*(pNodo).padre == \text{NULL}$ then	$\mathcal{O}(1)$
56: $*(pNodo).izq).padre \leftarrow \text{NULL}$	$\mathcal{O}(1)$

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

114: **end while**

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

Siendo n la cantidad de nodos.

Nuevamente ignoraremos los condicionales y asignaciones (dado que son constantes), y nos centraremos en la creación de "camino", y los posteriores tres ciclos.

-La creación de "camino", como hemos visto toma $\mathcal{O}(\log_2 n)$, dado que es crear un array con (a lo sumo) $\log_2 n + 1$ posiciones (esto es el camino recorrido, a rebalancear).

-El primer ciclo consiste en buscar el elemento a borrar, dado que es una búsqueda en un AVL, esto es $\mathcal{O}(\log_2 n)$.

-El segundo ciclo sucede sólo cuando el elemento a borrar tiene dos subárboles hijos distintos de NULL, esto consiste en buscar el sucesor in-order (es decir, bajar un nodo a la derecha, y luego bajar lo máximo posible hacia la izquierda. Así se encuentra el siguiente "inmediato"). Dado que es una búsqueda, y se empieza a descender desde el nodo a borrar (en peor caso, se empieza desde la raíz), esto toma $\mathcal{O}(\log_2 n)$. Cabe aclarar que éste ciclo no siempre se ejecuta, pero dado que en los demás casos la complejidad es de $\mathcal{O}(1)$, podemos asumir que dado el caso que haya sido, estará acotado por la complejidad del peor, osea éste.

-El tercer ciclo consiste en recorrer los elementos de "camino" de atrás hacia adelante (e ir rotando según corresponda), esto toma $\mathcal{O}(\log_2 n)$.

-Finalmente, la complejidad total es la suma de todas estas complejidades parciales: $\mathcal{O}(\log_2 n) + \mathcal{O}(\log_2 n) + \mathcal{O}(\log_2 n) + \mathcal{O}(\log_2 n) = 4 * \mathcal{O}(\log_2 n) = \mathcal{O}(\log_2 n)$

IVACÍO?(in d : diccRapido(α, β)) $\rightarrow res$: bool

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

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

ICLAVEMAX(in d : diccRapido(α, β)) $\rightarrow res$: α

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

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

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

ICLAVES(in d : diccRapido(α, β)) $\rightarrow res$: itClave

1: $res \leftarrow \text{CREARIT}(d)$	$\mathcal{O}(1)$
---------------------------------------	------------------

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

ICREARIT(in d : diccRapido(α, β)) $\rightarrow res$: itClave

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

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

IHAYMAS?(in it: itClave) $\rightarrow res : \text{bool}$

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

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

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

1: $res \leftarrow *(it.3).clave$	$\mathcal{O}(1)$
-----------------------------------	------------------

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

IAVANZAR(in/out it: itClave)

1: $it.1 \leftarrow it.1 + 1$	$\mathcal{O}(1)$
2: var itNodosNivelActual $\leftarrow \text{CREARIT}(\text{DAMENODOS}(it.4, 1, it.0))$	$\mathcal{O}(n)$
3: var bAvanzar:bool $\leftarrow \text{true}$	$\mathcal{O}(1)$
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 $\leftarrow \text{false}$	$\mathcal{O}(1)$
8: else	
9: end if	
10: end while	
11: if HAYSIGUIENTE?(itNodosNivelActual) then	$\mathcal{O}(1)$
12: $it.3 \leftarrow \text{SIGUIENTE}(itNodosNivelActual)$	$\mathcal{O}(1)$
13: else	
14: $it.0 \leftarrow it.0 + 1$	$\mathcal{O}(1)$
15: $it.3 \leftarrow \text{SIGUIENTE}(\text{CREARIT}(\text{DameNodos}(it.4, 1, it.0)))$	$\mathcal{O}(n)$
16: end if	

Complejidad: $\mathcal{O}(n^2)$

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) $\rightarrow res : \text{Conj}(\text{nodo})$

1: $res \leftarrow \text{VACÍO}()$	$\mathcal{O}(1)$
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: $\mathcal{O}(n)$

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

1: if FACTORDESBALANCE(p) < 1 then	$\mathcal{O}(1)$
2: if FACTORDESBALANCE(*(p).der) > 1 then	$\mathcal{O}(1)$
3: $res \leftarrow \text{ROTARDOBLEIZQ}(p)$	$\mathcal{O}(1)$
4: else	
5: $res \leftarrow \text{ROTARSIMPLEIZQ}(p)$	$\mathcal{O}(1)$
6: end if	
7: else	
8: if FACTORDESBALANCE(*(p).izq) < 1 then	$\mathcal{O}(1)$
9: $res \leftarrow \text{ROTARDOBLEDER}(p)$	$\mathcal{O}(1)$
10: else	
11: $res \leftarrow \text{ROTARSIMPLEDER}(p)$	$\mathcal{O}(1)$
12: end if	
13: end if	

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

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

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 != NULL then	
8: if *(r).clave == (*(padre).izq).clave then	$\mathcal{O}(1)$
9: *(padre).izq \leftarrow r2	$\mathcal{O}(1)$

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: $\mathcal{O}(1)$