Algoritmos y Estructuras de Datos II

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

Trabajo Práctico Número 2

DCNet

Grupo: 21

Integrante	LU	Correo electrónico
Alvarez, Lautaro Leonel	268/14	lautarolalvarez@gmail.com
Maddonni, Axel Ezequiel	200/14	axel.maddonni@gmail.com
Thibeault, Gabriel Eric	114/13	grojo94@hotmail.com
Vigali, Leandro Ezequiel	951/12	leandrovigali@yahoo.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo Red

Interfaz

usa: CONJ(α), ITCONJ(α), LISTA(α), ITLISTA(α).

se explica con: RED.

géneros: `red`.

Operaciones de Red

COMPUTADORAS(**in** r: `red`) $\rightarrow$ res : `conj(hostname)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ dameHostnames(computadoras(r))}
Complejidad: O(n)
Descripción: devuelve el conjunto de las computadoras.
Aliasing: res se devuelve por copia.

CONECTADAS?(**in** r: `red`, **in** $c1$: `hostname`, **in** $c2$: `hostname`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$c1$, $c2 \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ conectadas?(r, dameCompu($c1$), dameCompu($c2$))}
Complejidad: O(n*L)
Descripción: indica si las computadoras estan conectadas por alguna de sus interfaces.

INTERFAZUSADA(**in** r: `red`, **in** $c1$: `hostname`, **in** $c2$: `hostname`) $\rightarrow$ res : `interfaz`
Pre $\equiv$ {conectadas?(r, dameCompu($c1$), dameCompu($c2$))}
Post $\equiv$ {$res =_{obs}$ interfazUsada(r, dameCompu($c1$), dameCompu($c2$))}
Complejidad: O(n*L)
Descripción: devuelve la interfaz por la cual estan conectadas c1 y c2.

INICIARRED() $\rightarrow$ res : `red`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ iniciarRed()}
Complejidad: O(1)
Descripción: crear una nueva Red.

AGREGARCOMPU(**in/out** r: `red`, **in** $c1$: `compu`)
Pre $\equiv$ {$r = r_0 \wedge$ ($\forall$ c: compu) $c \in$ computadoras(r_0) $\Rightarrow$ ip(c) $\neq c1$}
Post $\equiv$ {$r =_{obs}$ agregarComputadora(r_0, $c1$)}
Complejidad: O(L+i) i=cantidad de interfaces
Descripción: agregar una computadora a la Red.
Aliasing: la computadora se agrega por copia.

CONECTAR(**in** r: `red`, **in** c_1: `hostname`, **in** i_1: `interfaz` **in** c_2: `hostname`, **in** i_2: `interfaz`)
Pre $\equiv$ {$r = r_0 \wedge c_1, c_2 \in$ dameHostnames(computadoras(r)) $\wedge$ $c_1 \neq c_2 \wedge$ $\neg$conectadas?(r, dameCompu(c_1), dameCompu(c_2)) $\wedge$ $\neg$usaInterfaz?(r, dameCompu(c_1), i_1) $\wedge$ $\neg$usaInterfaz?(r, dameCompu(c_2), i_2) $\wedge$ $i_1 \in$ dameCompu(c_1).interfaces $\wedge$ $i_2 \in$ dameCompu(c_2).interfaces}
Post $\equiv$ {$r =_{obs}$ (conectar(r_0, dameCompu(c_1), i_1, dameCompu(c_2), i_2)}
Descripción: conectar dos computadoras de la red.

VECINOS(**in** r: `red`, **in** c: `hostname`) $\rightarrow$ res : `conj(hostname)`
Pre $\equiv$ {$c \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ dameHostnames(vecinos(r, dameCompu(c)))}
Complejidad: O(n*L)
Descripción: da el conjunto de computadoras vecinas.

Aliasing: el conjunto se devuelve por copia.

USAINTERFAZ?(**in** r : `red`, **in** c : `hostname`, **in** i : `interfaz`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$c \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ usaInterfaz?(r, dameCompu(c), i)}
Complejidad: O(n)
Descripción: indica si la interfaz está siendo utilizada.

CAMINOSMINIMOS(**in** r : `red`, **in** c_1 : `hostname`, **in** c_2 : `hostname`) $\rightarrow$ res : `conj(lista(hostname))`
Pre $\equiv$ {$c1,\ c2 \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ dameCaminosdeHostnames(caminosMinimos(r, dameCompu(c_1), dameCompu(c_2)))}
Complejidad: O(n*L)
Descripción: devuelve los conjuntos de caminos minimos entre las computadoras ingresadas.
Aliasing: res se devuelve por copia.

HAYCAMINO?(**in** r : `red`, **in** c_1 : `hostname`, **in** c_2 : `hostname`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$c1,\ c2 \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ hayCamino?(r, dameCompu(c_1), dameCompu(c_2))}
Complejidad: O(n*n)
Descripción: indica si las computadoras son alcanzables mediante algún camino.

$\bullet == \bullet$(**in** r_1 : `red`, **in** r_2 : `red`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ ($r_1 =_{obs} r_2$}
Complejidad: O(n*n * (L + n*n + m) + n*m*m)
Descripción: indica si dos redes son iguales.

COPIAR(**in** r : `red`) $\rightarrow$ res : `red`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ r}
Descripción: copia la red.
Aliasing: res se devuelve por copia

donde:
`hostname` es `string`,
`interfaz` es `nat`,
`compu` es `tupla`<ip: `hostname`, interfaces: `conj(interfaz)`>.

Especificación de las operaciones auxiliares utilizadas en la interfaz (no exportadas)

TAD RED EXTENDIDA

extiende RED

otras operaciones

damehostnames	: conj(compu) $\longrightarrow$ conj(hostname)	
dameCompu	: red r $\times$ hostname s $\longrightarrow$ compu	{$s \in$ hostnames(r)}
auxDameCompu	: red r $\times$ hostname s $\times$ conj(compu) cc $\longrightarrow$ compu	{$s \in$ hostnames(r) $\wedge$ $cc \subset$ computadoras(r)}
dameCaminosDeHostnames	: conj(secu(compu)) $\longrightarrow$ conj(secu(hostname))	
dameSecuDeHostnames	: secu(compu) $\longrightarrow$ secu(hostname)	

axiomas $\forall\ r$: red, $\forall\ cc$: conj(compu), $\forall\ s$: hostname, $\forall\ cs$: conj(secu(compu)), $\forall\ secu$: secu(compu)

dameHostnames(cc) $\equiv$ **if** vacio?(cc) **then**
$\emptyset$
else
Ag(ip(dameUno(cc)), dameHostnames(sinUno(cc)))
fi

dameCompu(r, s) $\equiv$ auxDameCompu(r, s, computadoras(r))

auxDameCompu(r, s, cc) $\equiv$ **if** ip(dameUno(cc)) $= s$ **then**
dameUno(cc)
else
auxDameCompu(r, s, sinUno(cc))
fi

dameCaminosDeHostnames(cs) $\equiv$ **if** vacio?(cs) **then**
$\emptyset$
else
Ag(dameSecuDeHostnames(dameUno(cs)),
dameCaminosDeHostnames(sinUno(cs)))
fi

dameSecuDeHostnames($secu$) $\equiv$ **if** vacia?($secu$) **then**
<>
else
ip(prim($secu$)) $\bullet$ dameSecuDeHostnames(fin($secu$))
fi

Fin TAD

Representación

`red` **se representa con** `estr_red`

donde:

`estr_red` es `dicc(hostname, datos)`

donde `datos` es `tupla`(*interfaces*: `conj(interfaz)`
conexiones: `dicc(interfaz, hostname)`
alcanzables: `dicc(dest: hostname, caminos: conj(lista(hostname)))`)

`hostname` es `string`, `interfaz` es `nat`.

\\ Rep en Castellano:
Para cada computadora:
1: Las interfaces usadas pertenecen al conjunto de interfaces de la compu.
2: Los vecinos perteneces a las computadoras de la red.
3: Los vecinos son distintos a la compu actual.
4: Los vecinos no se repiten.
5: Las conexiones son bidireccionales.
6: Los alcanzables pertenecen a las computadoras de la red.
7: Los alcanzables son distintos a la actual.
8: Los alcanzables tienen un camino válido hacia ellos desde la actual.
9: Para cada alcanzable, el conjunto de camiinos válidos no es vacío.
10: Todos los caminos en el diccionario alcanzables son válidos.
11: Los caminos son mínimos.
12: Están todos los mínimos.

Rep : estr_red $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$ ($\forall$ c: hostname, $c \in$ claves(e)) (
\\ *1* claves(obtener(e, c).conexiones) $\subseteq$ obtener(e, c).interfaces $\wedge$
($\forall$ i: interfaz, $i \in$ claves(obtener(e, c).conexiones)) (
\\ *2* obtener(obtener(e, c).conexiones, i) $\in$ claves(e) $\wedge$
\\ *3* obtener(obtener(e, c).conexiones, i) $\neq c$ $\wedge$
\\ *4* ($\neg$ $\exists$ i': interfaz, $i' \in$ claves(obtener(e, c).conexiones), $i \neq i'$) obtener(obtener(e, c).conexiones, i) == obtener(obtener(e, c).conexiones, i') $\wedge$
\\ *5* ($\forall$ h: hostname) (h == obtener(obtener(e, c).conexiones, i) $\Rightarrow$ ($\exists$ i':int) obtener(obtener(e, h).conexiones, i') == c)) $\wedge$
\\ *6* claves(obtener(e, c).alcanzables $\subseteq$ claves(e) $\wedge$
($\forall$ a: hostname, $a \in$ claves(obtener(e, c).alcanzables) (
\\ *7* $a \neq c$ $\wedge$
\\ *8* ($\exists$ s: secu(hostname)) esCaminoVálido(c, a, s) $\wedge$
\\ *9* #obtener(obtener(e, c).alcanzables, a) >0 $\wedge_L$
($\forall$ *camino*: secu(hostname), *camino* $\in$ obtener(obtener(e, c).alcanzables, a) (
\\ *10* esCaminoVálido(c, a, *camino*) $\wedge$
\\ *11* $\neg$($\exists$ *camino*$'$: secu(hostname), *camino* $\neq$ *camino*$'$, esCaminoVálido(c, a, *camino*$'$)) long(*camino*$'$) <long(*camino*) $\wedge$
\\ *12* $\neg$($\exists$ *camino*$'$: secu(hostname), *camino* $\neq$ *camino*$'$, esCaminoVálido(c, a, *camino*$'$), long(*camino* == long(*camino*$'$)) (*camino*$'$ $\notin$ obtener(obtener(e, c).alcanzables, a))
))

La abreviatura *esCaminoValido* usada en el Rep se debe leer: (no son funciones, son abreviaturas para hacer más fácil la lectura)

esCaminoValido (*orig*, *dest*, *secu*) $\equiv$ (prim(*secu*) == *orig* $\wedge$
($\forall$ i: nat, $0 < i <$ long(*secu*)) esVecino (*secu*[i], *secu*[$i+1$]) $\wedge$
secu [long(*secu*)-1] == *dest* $\wedge$
sinRepetidos(*secu*))

Con *esVecino* ($h1$, $h2$) $\equiv$ ($\exists$ i: interfaz) $h2$ == obtener (obtener(e, $h1$).conexiones, i)

Abs : estr_red e $\longrightarrow$ red {Rep(e)}

Abs(e) $\equiv$ r | computadoras (r) = dameComputadoras(e) $\wedge_L$
($\forall$ $c1$, $c2$: compu, $c1$, $c2 \in$ computadoras(r)) conectadas?(r, $c1$, $c2$) = ($\exists$ i: interfaz) ($c2$.ip = obtener(obtener(e, $c1$.ip).conexiones, i)) $\wedge_L$
interfazUsada(r, $c1$, $c2$) = buscarClave (obtener(e, $c1$.ip).conexiones, $c2$.ip)

Especificación de las funciones auxiliares utilizadas en abs

dameComputadoras : dicc(hostname;X) $\longrightarrow$ conj(computadoras)

auxDameComputadoras : dicc(hostname;X) $\times$ conj(hostname) $\longrightarrow$ conj(computadoras)

buscarClave : dicc(interfaz;hostname) $\times$ hostname $\longrightarrow$ interfaz

auxBuscarClave : dicc(interfaz;hostname) $\times$ hostname $\times$ conj(interfaz) $\longrightarrow$ interfaz

axiomas $\forall$ e: dicc(hostname, X), $\forall$ d: dicc(interfaz, hostname), $\forall$ cc: conj(hostname), $\forall$ ci: conj(interfaz), $\forall$ h: hostname

dameComputadoras(e) $\equiv$ auxDameComputadoras(e, claves(e)

auxDameComputadoras(e, cc) $\equiv$ **if** $\emptyset?(cc)$ **then**
$\emptyset$
else
Ag(<dameUno(cc), obtener(e, dameUno(cc)).interfaces>, auxDameComputadoras(e, sinUno(cc)))
fi

buscarClave(d, h) $\equiv$ auxBuscarClave(d, h, claves(d))

auxBuscarClave(d, h, ci) $\equiv$ **if** obtener(d, dameUno(cc)) $= h$ **then**
dameUno(cc)
else
auxBuscarClave(d, h, sinUno(ci))
fi

Algoritmos

Algorithm 1 Implementación de Computadoras

```
function iComputadoras(in r: estr_red)→ res: conj(hostname)
    it ← crearIt(r)                                                          ▷ O(1)
    res ← Vacio()                                  ▷ conjunto                ▷ O(1)
    while HaySiguiente(it) do        ▷ Guarda: O(1)   ▷ El ciclo se ejecuta n veces   ▷ O(n)
        Agregar(res, SiguienteClave(it))                                     ▷ O(1)
        Avanzar(it)                                                          ▷ O(1)
    end while
end function                                                                 ▷ O(n)
```

Algorithm 2 Implementación de Conectadas?

```
function iConectadas?(in r: estr_red, in c1: hostname, in c2: hostname)→ res: bool
    it ← CrearIt(Significado(r,c1).conexiones)                               ▷ O(n)
    res ← FALSE                                                              ▷ O(1)
    while HaySiguiente(it) && ¬res do ▷ Guarda: O(1) ▷ El ciclo se ejecuta a lo sumo n-1 veces ▷ O(n)
        if SiguienteClave(it)==c2 then                                       ▷ O(L)
            res ← TRUE
        end if
        Avanzar(it)                                                          ▷ O(1)
    end while
end function                                                                 ▷ O(n*L)
```

Algorithm 3 Implementación de InterfazUsada

```
function iInterfazUsada(in r: estr_red, in c1: hostname, in c2: hostname)→ res: interfaz
    it ← CrearIt(significado(r,c1).conexiones)                               ▷ O(n)
    while HaySiguiente(it) do     ▷ Guarda: O(1)   ▷ El ciclo se ejecuta a lo sumo n-1 veces   ▷ O(n)
        if SiguienteSignificado(it)==c2 then                                 ▷ O(L)
            res ← SiguienteClave(it)                ▷ nat por copia          ▷ O(copiar(nat))
        end if
        Avanzar(it)                                                          ▷ O(1)
    end while
end function                                                                 ▷ O(n*L)
```

Algorithm 4 Implementación de IniciarRed

```
function iIniciarRed()→ res: estr_red
    res ← Vacio()                                  ▷ Diccionario                ▷ O(1)
end function                                                                    ▷ O(1)
```

Algorithm 5 Implementación de AgregarCompu

```
function iAgregarCompu(inout r: estr_red, in c1: compu)
    nuevoDiccVacio ← Vacio()                       ▷ Diccionario                ▷ O(1)
    DefinirRapido(r, c1.ip, tupla(Copiar(c1.interfaces), nuevoDiccVacio, nuevoDiccVacio))   ▷
O(Copiar(sL) + Copiar(conj(interfaz)) con sL=string de largo L
end function                                                                    ▷ O(L + i) con i=cantidad de interfaces
```

Algorithm 6 Implementación de Conectar

```
function iConectar(inout r: estr_red, in c1: hostname, int i1:interfaz, in c2: hostname, in i2:interfaz)
\\ Actualizo conexiones de ambas
    DefinirRapido(Significado(r, c1).conexiones, i1, c2)        ▷ O(n) + O(L) + O(copiar(nat))
    DefinirRapido(Significado(r, c2).conexiones, i2, c1)        ▷ O(n) + O(L) + O(copiar(nat))
\\ Actualizo caminos de ambas
    ActualizarCaminos(r, c1, c2)
    ActualizarCaminos(r, c2, c1)
\\ Creo conjunto con los actualizados hasta el momento
    actualizados ← Vacio()                                      ▷ conjunto
    AgregarRapido(actualizados, c1)
    AgregarRapido(actualizados, c2)
\\ Actualizo caminos del resto de la Red por recursion
    ActualizarVecinos(r, c1, actualizados)
end function
```

Algorithm 7 Implementación de función auxiliar Actualizar Caminos

```
function IACTUALIZARCAMINOS(inout r: estr_red, in c1: hostname, in c2: hostname)
\\ Actualiza los caminos de c1 con los de c2
\\ Recorro los alcanzables de c2
    itAlcanzables2 ← crearIt(Significado(r, c2).alcanzables)                                   ▷ O(1)
    while (HaySiguiente(itAlcanzables2) do          ▷ se ejecuta a lo sumo n-1 veces           ▷ O(n)
\\ Recorro alcanzables de c1
        itAlcanzables1 ← crearIt(Significado(r, c1).alcanzables)                               ▷ O(1)
        while (HaySiguiente(itAlcanzables1)) do      ▷ se ejecuta a lo sumo n-1 veces           ▷ O(n)
            if (SiguienteClave(itAlcanzables2) == SiguienteClave(itAlcanzables1)) then  ▷ O(n) + O(L)
\\ El alcanzable ya estaba, me fijo que caminos son más cortos
                itCaminos ← crearIt (SiguienteSignificado(itAlcanzables2))                     ▷ O(1)
                camino2 ← Siguiente(itCaminos)          ▷ camino minimo del c2                 ▷ O(1)
                itCaminos ← crearIt (SiguienteSignificado(itAlcanzables1))                     ▷ O(1)
                camino1 ← Siguiente(itCaminos)          ▷ camino minimo del c1                 ▷ O(1)
                if (longitud(camino1) >longitud(camino2)) then      ▷ cada camino tiene a lo sumo n
elementos                                                                                      ▷ O(n)
\\ Los caminos nuevos son mas cortos, borro los que estÃ¡n y copio los nuevos
                    Borrar(Significado(r, c1).alcanzables, SiguienteClave(itAlcanzables1)) ▷ O(n) + O(n*L)
\\ Nuevo alcanzable: me copio los caminos agregando c1 al principio
                    itCaminos ← crearIt (SiguienteSignificado(itAlcanzables2))                 ▷ O(1)
                    caminos ← Vacio()  ▷ conjunto donde voy a guardar los caminos modificados  ▷ O(1)
                    while  (HaySiguiente(itCaminos)) do
                        nuevoCamino ← copy(Siguiente(itCaminos))  ▷ copio el camino que voy a modificar
                        AgregarAdelante(nuevoCamino, c1)
                        AgregarRapido (caminos, nuevoCamino)
                        Avanzar(itCaminos)
                    end while
\\ agrego el nuevo alcanzable con el camino
                    DefinirRapido(Significado(r,c1).alcanzables, SiguienteClave(itAlcanzables2), caminos)
                else
                    if (longitud(camino1) == longitud(camino2)) then
\\  Tengo que agregar los nuevos caminos (modificados) al conjunto de caminos actual
                        itCaminos ← crearIt(SiguienteSignificado(itAlcanzables2))
                        while (HaySiguiente(itCaminos)) do
                            nuevoCamino ← copy(Siguiente(itCaminos))        ▷ copio el camino que voy a
modificar
                            AgregarAdelante(nuevoCamino, c1)
                            Agregar(SiguienteSignificado(itAlcanzables1), nuevoCamino)
                            Avanzar(itCaminos)
                        end while
                    end if
                end if
            else
\\ Nuevo alcanzable : me copio los caminos agregando c1 al principio
                itCaminos ← crearIt (SiguienteSignificado(itAlcanzables2))
                caminos ← Vacio()                ▷ conjunto donde voy a guardar los caminos modificados
                while  (HaySiguiente(itCaminos)) do
                    nuevoCamino ← copy(Siguiente(itCaminos))      ▷ copio el camino que voy a modificar
                    AgregarAdelante(nuevoCamino, c1)
                    AgregarRapido (caminos, nuevoCamino)
                    Avanzar(itCaminos)
                end while
                DefinirRapido(Significado(r, c1).alcanzables, SiguienteClave(itAlcanzables2), caminos)
            end if
            Avanzar(itAlcanzables1)
        end while
        Avanzar(itAlcanzables2)
    end while
end function
```

Algorithm 8 Implementación de función auxiliar Actualizar Vecinos

```
function IACTUALIZARVECINOS(inout r: estr_red, in c1: hostname, in conj(hostname) actualizados)
\\ Actualiza los caminos de los vecinos de C, y luego hace recursion para los vecinos de los vecinos.
    itVecinos ← crearIt(Significado(r, c1).conexiones)
    while ( HaySiguiente(itVecinos) ) do
\\ Si todavÃa no fue actualizado, lo actualizo y hago recursión sobre los vecinos.
        if (SiguienteClave(itVecinos) ∉ actualizados) then
            ActualizarCaminos(r, SiguienteClave(itVecinos), c )
            AgregarRapido (actualizados, SiguienteClave(itVecinos))
            ActualizarVecinos(r, SiguienteClave(itVecinos), actualizados)
        end if
        Avanzar(itVecinos)
    end while
end function
```

Algorithm 9 Implementación de Vecinos

```
function IVECINOS(inout r: estr_red, in c1: hostname)→ res: conj(hostname)
    it ← CrearIt(Significado(r,c1).conexiones)                          ▷ O(1) + O(n)
    res ← Vacio()                                  ▷ Conjunto           ▷ O(1)
    while HaySiguiente(it) do    ▷ Guarda: O(1)   ▷ El ciclo se ejecuta a lo sumo n-1 veces   ▷ O(n)
        AgregarRapido(res, SiguienteSignificado(it))                    ▷ O(L)
        Avanzar(it)                                                     ▷ O(1)
    end while
end function                                                            ▷ O(n*L)
```

Algorithm 10 Implementación de UsaInterfaz

```
function IUSAINTERFAZ(in r: estr_red, in c: hostname, in i: interfaz)→ res: bool
    res ← Definido?(Significado(r,c).conexiones,i)                      ▷ O(comparar(nat)*n)
end function                                                            ▷ O(n)
```

Algorithm 11 Implementación de CaminosMinimos

```
function ICAMINOSMINIMOS(in r: estr_red, in c1: hostname, in c2: hostname)→ res:
conj(lista(hostname))
    itCaminos ← crearIt(Significado(Significado(r,c1).alcanzables, c2))   ▷ O(1) + O(L*n)
    res ← Vacio()                                  ▷ Conjunto           ▷ O(1)
    while HaySiguiente(itCaminos) do
        AgregarRapido(res, Siguiente(itCaminos))                        ▷ O(1)
        Avanzar(itCaminos)                                              ▷ O(1)
    end while
end function                                                            ▷ O(n*L)
```

Algorithm 12 Implementación de HayCamino?

```
function IHAYCAMINO?(in r: estr_red, in c1: hostname, in c2: hostname)→ res: bool
    res ← Definido?(Significado(r,c1).alcanzables, c2)                  ▷ O(n*n)
end function                                                            ▷ O(n*n)
```

Algorithm 13 Implementación de ==

```
function IIGUALDAD(in r1: estr_red, in r2: estr_red)→ res: bool
    res ← TRUE                                                              ▷ O(1)
    if ¬(#Claves(r1)==#Claves(r2)) then                                     ▷ O(comparar(nat))
\\ Si la cantidad de claves son distintas =>las redes son distintas
        res ← FALSE                                                         ▷ O(1)
    else
        itRed1 ← CrearIt(r1)                                                ▷ O(1)
        while HaySiguiente(itRed1) && res do    ▷ Guarda: O(1)   ▷ Se ejecuta n veces   ▷ O(n)
\\ Recorro la red 1 y me fijo para cada una de sus computadoras
            if ¬(Definido?(r2, SiguienteClave(itRed1)) then                 ▷ O(L*n)
\\ Si no estÃ¡ definido su hostname en la red 2 =>las redes son distintas
                res ← FALSE                                                 ▷ O(1)
            else
                Compu2 ← Significado(r2,SiguienteClave(itRed1))             ▷ O(L*n)
                Compu1 ← SiguienteSignificado(itRed1)                       ▷ O(1)
\\ Tomo las computadoras de red 1 y red 2 con el mismo hostname y las comparo
                if ¬(Comp1.interfaces == Comp2.interfaces) then     ▷ O(m*m) con m=cantidad de
interfaces
\\ Si sus interfaces son distintas =>las redes son distintas
                    res ← FALSE                                             ▷ O(1)
                end if
                if ¬(Comp1.conexiones == Comp2.conexiones) then
\\ Si sus conexiones son distintas =>las redes son distintas
                    res ← FALSE                                             ▷ O(1)
                end if
                if ¬(#Claves(Compu1.alcanzables)==#Claves(Compu2.alcanzables)) then
\\ Si sus cantidades de alcanzables son distintas =>las redes son distintas
                    res ← FALSE                                             ▷ O(1)
                else
                    itAlc1 ← CrearIt(Compu1.alcanzables)                    ▷ O(1)
                    while HaySiguiente(itAlc1) && res do   ▷ se ejecuta a lo sumo n−1 veces   ▷ O(n)
\\ Para cada alcanzable de la computadora de la red 1
                        if ¬(Definido?(Comp2.alcanzables, SiguienteClave(itAlc1))) then   ▷ O(m)
\\ Si no está definida en los alcanzables de la compu de la red 2 =>las redes son distintas
                            res ← FALSE
                        else
                            Caminos1 ← SiguienteSignificado(itAlc1)         ▷ O(1)
                            Caminos2 ← Significado(Comp2.alcanzables, itAlc1)   ▷ O(n)
\\ Me guardo los 2 conjuntos de caminos (de la compu de la red 1 y la de la red 2)
```

```
                    if ¬(Longitud(Caminos1) == Longitud(Caminos2)) then   ▷ O(comparar(nat))
\\ Si sus cantidades son distintas =>las redes son distintas
                        res ← FALSE
                    else
                        itCaminos1 ← CrearIt(Caminos1)                    ▷ O(1)
                        while HaySiguiente(itCaminos1) && res do
\\ Para cada camino en el conjunto de caminos de la compu de la red 1
\\ Recorro los caminos de la compu de la red 2
                            itCaminos2 ← CrearIt(Caminos2)                ▷ O(1)
                            noEncontro ← TRUE                             ▷ O(1)
                            while HaySiguiente(itCaminos2) && noEncontro do
\\ Busco que el camino de la compu de la red 1 esté en la compu de la red 2
                                if Siguiente(itCaminos2) == Siguiente(ItCaminos1) then
                                    noEncontro ← FALSE
                                end if
                                Avanzar(itCaminos2)                       ▷ O(1)
                            end while
                            if noEncontro then                            ▷ O(1)
\\ Si no encontró alguno =>las redes son distintas
                                res ← FALSE                               ▷ O(1)
                            end if
                            Avanzar(itCaminos1)                           ▷ O(1)
                        end while
                    end if
                end if
                Avanzar(itAlc1)                                           ▷ O(1)
            end while
        end if
    end if
    Avanzar(itRed1)                                                       ▷ O(1)
  end while
 end if
end function                                          ▷ O(n*n * ( L + n*n + m ) + n*m*m)
```

Algorithm 14 Implementación de Copiar

```
function iCopiar(in r: estr_red)→ res: red
    res ← IniciarRed()                                                          ▷ O(1)
\\ Crea una red vacia.
    itRed ← CrearIt(r)                                                          ▷ O(1)
    while HaySiguiente(itRed) do          ▷ O(1)        ▷ se ejecuta n veces    ▷ O(n)
\\ Para cada computadora en la red original.
        copiaAlcanzables ← Vacio()                        ▷ diccionario         ▷ O(1)
\\ Inicia los alcanzables en vacio.
        itAlcanzables ← CrearIt(SiguienteSignificado(itRed).alcanzables)        ▷ O(1)
        while HaySiguiente(itAlcanzables) do  ▷ Guarda: O(1)  ▷ se ejecuta a lo sumo n veces  ▷ O(n)
\\ Para cada conjunto de caminos mínimos (cada destino).
            copiaCaminos ← Vacia()                        ▷ lista               ▷ O(1)
\\ Inicia el conjunto de caminos mínimos como vacío.
            itCaminos ← CrearIt(SiguienteSignificado(itAlcanzables))            ▷ O(1)
            while HaySiguiente(itCaminos) do
\\ Para cada camino en el conjunto original.
                AgregarAdelante(copiaCaminos, Siguiente(itCaminos))             ▷ O(copiar(camino))
\\ Copia el camino original y lo agrega adelante del conjunto de caminos mínimos.
                Avanzar(itCaminos)                                              ▷ O(1)
            end while
            Definir(copiaAlcanzables, SiguienteClave(itAlcanzables), copiaCaminos)
\\ Define el destino y sus caminos mínimos en la copia de alcanzables.
            Avanzar(itAlcanzables)                                              ▷ O(1)
        end while
        Definir(res, SiguienteClave(itRed), Tupla(Copiar(SiguienteSignificado(itRed).interfaces), Copiar(SiguienteSignificado(itRed).conexiones), copiaAlcanzables))
\\ Define la copia de la computadora con los campos antes copiados.
        Avanzar(itRed)                                                          ▷ O(1)
    end while
end function
```

2. Módulo DCNet

Interfaz

usa: RED, CONJ(α), ITCONJ(α), LISTA(α), ITLISTA(α), DICC$_{UNIV}$(κ, σ), DICC$_{LOG}$(κ, σ), COLA$_{LOG}$(α).

se explica con: DCNET.

géneros: `dcnet`.

Operaciones de DCNet

RED(**in** d: `dcnet`) $\rightarrow$ res : `red`
Pre $\equiv$ {true}
Post $\equiv$ {alias($res =_{obs}$ red(d))}
Complejidad: O()
Descripción: devuelve la red asociada.
Aliasing: res no es modificable.

CAMINORECORRIDO(**in** d: `dcnet`, **in** p: `IDpaquete`) $\rightarrow$ res : `lista(hostname)`
Pre $\equiv$ {IDpaqueteEnTransito?(d, p)}
Post $\equiv$ {$res =_{obs}$ dameSecuDeHostnames(caminoRecorrido(d, damePaquete(p)))}
Complejidad: O()
Descripción: devuelve el camino recorrido desde el origen hasta el actual.
Aliasing: res se devuelve por copia.

CANTIDADENVIADOS(**in** d: `dcnet`, **in** c: `hostname`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$c \in$ dameHostnames(computadoras(red(d)))}
Post $\equiv$ {$res =_{obs}$ cantidadEnviados(d, dameCompu(c))}
Complejidad: O(L)
Descripción: devuelve la cantidad de paquetes enviados por la computadora.

ENESPERA(**in** d: `dcnet`, **in** c: `hostname`) $\rightarrow$ res : `conj(paquete)`
Pre $\equiv$ {$c \in$ dameHostnames(computadoras(red(d)))}
Post $\equiv$ {alias($res =_{obs}$ enEspera(d, dameCompu(c)))}
Complejidad: O(L)
Descripción: devuelve los paquetes en la cola de la computadora.
Aliasing: res no es modificable.

INICIARDCNET(**in** r: `red`) $\rightarrow$ res : `dcnet`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ iniciarDCNet(r)}
Complejidad: O()
Descripción: crea una nueva Dcnet.
Aliasing: la red se agrega por copia.

CREARPAQUETE(**in/out** d: `dcnet`, **in** p: `paquete`)
Pre $\equiv$ {$d = d_0 \wedge \neg($ ($\exists$ p': paquete) (paqueteEnTransito?(d, p') $\wedge$ id(p') = 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, p)}
Complejidad: O()
Descripción: agrega un paquete a la red.
Aliasing: el paquete se agrega por copia.

AvanzarSegundo(**in/out** d: `dcnet`)
Pre $\equiv \{d = d_0\}$
Post $\equiv \{d =_{obs}$ avanzarSegundo$(d_0)\}$
Complejidad: O()
Descripción: realiza los movimientos de paquetes correspondientes, aplicando los cambios necesarios a la dcnet.

PaqueteEnTransito?(**in** d: `dcnet`, **in** p: `IDpaquete`) $\rightarrow$ res : `bool`
Pre $\equiv \{$true$\}$
Post $\equiv \{res =_{obs}$ IDpaqueteEnTransito?$(d, p)\}$
Complejidad: O()
Descripción: indica si el paquete esta en alguna de las colas dado el ID.

LaQueMasEnvio(**in** d: `dcnet`) $\rightarrow$ res : `hostname`
Pre $\equiv \{$true$\}$
Post $\equiv \{res =_{obs}$ laQueMasEnvio(d).ip$\}$
Complejidad: O()
Descripción: devuelve la computadora que más paquetes envió.
Aliasing: res se devuelve por copia.

$\bullet = \bullet$(**in** d_1 : `dcnet`, **in** d_2 : `dcnet`) $\rightarrow$ res : `bool`
Pre $\equiv \{$true$\}$
Post $\equiv \{res =_{obs} (d_1 =_{obs} d_2\}$
Complejidad: O()
Descripción: indica si dos dcnet son iguales.

donde:
`hostname` es `string`,
`interfaz` es `nat`,
`IDpaquete` es `nat`,
`compu` es `tupla`<ip: `hostname`, interfaces: `conj(interfaz)`>,
`paquete` es `tupla`<id: `IDpaquete`, prioridad: `nat`, origen: `hostname`, destino: `hostname` >.

Especificación de las operaciones auxiliares utilizadas en la interfaz (no exportadas)

TAD DCNET EXTENDIDA

extiende DCNET

otras operaciones

damehostnames : conj(compu) $\longrightarrow$ conj(hostname)

dameCompu : dcnet d $\times$ hostname s $\longrightarrow$ compu $\{s \in \text{dameHostnames}(\text{computadoras}(\text{red}(d)))\}$

auxDameCompu : hostname s $\times$ conj(compu) cc $\longrightarrow$ compu

dameSecuDeHostnames : secu(compu) $\longrightarrow$ secu(hostname)

IDpaqueteEnTransito? : dcnet d $\times$ IDpaquete p $\longrightarrow$ bool

damePaquete : dcnet d $\times$ IDpaquete p $\longrightarrow$ paquete $\{\text{IDpaqueteEnTransito?}(d,p)\}$

dameIDpaquetes : conj(paquete) $\longrightarrow$ conj(IDpaquete)

axiomas $\forall$ d: dcnet, $\forall$ s: hostname, $\forall$ p: IDpaquete, $\forall$ cc: conj(compu), , $\forall$ $secu$: secu(compu), $\forall$ cp: conj(paquete),

dameHostnames(cc) $\equiv$ **if** vacio?(cc) **then** $\emptyset$ **else** Ag(ip(dameUno(cc)), dameHostnames(sinUno(cc))) **fi**

dameCompu(d, s) $\equiv$ auxDameCompu(s, computadoras(red((d)))

auxDameCompu(s, cc) $\equiv$ **if** ip(dameUno(cc)) $= s$ **then** dameUno(cc) **else** auxDameCompu(s, sinUno(cc)) **fi**

dameSecuDeHostnames($secu$) $\equiv$ **if** vacia?($secu$) **then** $<>$ **else** ip(prim($secu$)) $\bullet$ dameSecuDeHostnames(fin($secu$)) **fi**

IDpaqueteEnTransito?(d, p) $\equiv$ auxIDpaqueteEnTransito(d, computadoras(red(d)), p)

auxIDpaqueteEnTransito(d, cc, p) $\equiv$ **if** vacio?(cc) **then** false **else** **if** $p \in$ dameIDpaquetes(enEspera(dameUno(cc))) **then** true **else** auxIDpaqueteEnTransito(d, sinUno(cc), p) **fi** **fi**

dameIDpaquetes(cp) $\equiv$ **if** vacio?(cp) **then** $\emptyset$ **else** Ag(id(dameUno(cp)), dameIDpaquetes(sinUno(cp))) **fi**

Fin TAD

Representación

`dcnet` **se representa con** `estr_dcnet`

donde `estr_dcnet` es `tupla`(*red*: `red`
computadoras: `dicc(hostname, X)`
porHostname: $\texttt{dicc}_{UNIV}$ `(hostname, itDicc(hostname, X))`
conMasEnvios: `itDicc(hostname, X)`
caminos: `arreglo_dimensionable de arreglo_dimensionable de lista(hostname)`)

donde `X` es `tupla`(*indice*: `nat`
paquetes: `conj(paquete)`
cola: $\texttt{cola}_{LOG}$`(itConj(paquete))`
paqPorID: $\texttt{dicc}_{LOG}$ `(IDpaquete, itConj(paquete))`
cantEnvios: `nat`)

\\ REP en Castellano:
1: Las compus de Red son las compus de DCNet.
2: PorHostname y computadoras tienen el mismo conjunto de claves.
3: PorHostname permite acceder a los datos de todas las computadoras a través de iteradores.
4: Los indices de las computadoras van de 0 a n-1.
5: Los indices no se repiten.
6: ConMasEnvios es un interador a la computadora con mayor cant de envios.
7: La matriz de caminos es de n x n.
8: En la matriz caminos[i][j] se guarda uno de los caminos minimos de la red correspondiente al origen y destino correspondientes a los indices i, j, respectivamente. Si no hay, se guarda una lista vacia.
9: Las claves del diccionario paquetesPorID son los ID del conjunto paquetes.
10: El conjunto de paquetes y la cola de prioridad tienen el mismo tamano.
11: La cola ordena los paquetes por prioridad. (usando los observadores del TAD Cola de Prioridad Alaternativa adjunto).
Para todos los paquetes de una computadora:
12: El origen y el destino estan entre las computadoras de la dcnet.
13: El origen y el destino son distintos.
14: Hay un camino posible entre el origen y el destino.
15: La computadora actual esta en el camino minimo entre el origen y el destino.
16: El id es unico.
17: Son accesibles por el dicc usando su ID.
18: En la cola hay un iterador a cada paquete.

Rep : estr_dcnet $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
\\ *1* dameHostnames(computadoras(e.red)) = claves (e.computadoras) $\wedge$
\\ *2* claves (e.computadoras) = claves (e.porHostname) $\wedge$
\\ *3* ($\forall$ c: hostname, $c \in$ claves(e.porHostname)) (SiguienteClave(obtener(e.porHostname, c)) = c $\wedge$ SiguienteSignificado(obtener(e.porHostname, c)) = obtener(e.computadoras, c)) $\wedge$
($\forall$ c: hostname, $c \in$ claves(e.computadoras))
\\ *4* 0 < obtener(e.computadoras, c).indice < #claves(e.computadoras)-1 $\wedge$
\\ *5* $\neg$($\exists$ c': hostname, $c' \in$ claves(e.computadoras), $c \neq c'$) obtener(e.computadoras, c').indice = obtener(e.computadoras, c).indice $\wedge$
\\ *6* $\neg$($\exists$ c': hostname, $c' \in$ claves(e.computadoras), $c \neq c'$) obtener(e.computadoras, c').cantEnvios >SiguienteSignificado(e.conMasEnvios).cantEnvios $\wedge$
\\ *7* tam(e.caminos) = #claves(e.computadoras) $\wedge_L$ ($\forall$ i: nat, 0 < i < #claves(e.computadoras)-1) tam(e.caminos[i]) = #claves(e.computadoras) $\wedge$
\\ *8* ($\forall$ $c1, c2$: hostname, $c1, c2 \in$ claves(e.porHostname))
$\neg$ $\emptyset$? (caminosMinimos(e.red, dameCompu($c1$), dameCompu($c2$))) $\Rightarrow$ e.caminos[obtener(e.computadoras, $c1$).indice][obtener(e.computadoras, $c2$).indice] = dameUno(caminosMinimos(e.red, dameCompu($c1$), dameCompu($c2$))) $\wedge$
$\emptyset$? (caminosMinimos(e.red, dameCompu($c1$), dameCompu($c2$))) $\Rightarrow$ e.caminos[obtener(e.computadoras, $c1$).indice][obtener(e.computadoras, $c2$).indice] = Vacia() $\wedge$
($\forall$ c: hostname, $c \in$ claves(e.computadoras)) (
\\ *9* dameIDpaquetes(obtener(e.computadoras, c).paquetes) = claves(obtener(e.computadoras, c).paquetesPorID) $\wedge$
\\ *10* #(obtener(e.computadoras, c).paquetes) = #(obtener(e.computadoras, c).cola) $\wedge$
\\ *11* vacia?(obtener(e.computadoras, c).cola) = $\emptyset$?(obtener(e.computadoras, c).paquetes) $\wedge$
Siguiente(Π_2(proximo(obtener(e.computadoras, c).cola))) $\in$ obtener(e.computadoras, c).paquetes $\wedge$ $\neg$($\exists$ p': paquete, $p' \in$ obtener(e.computadoras, c).paquetes) p'.prioridad < Siguiente(Π_2(proximo(obtener(e.computadoras, c).cola))).prioridad $\wedge$
Π_1(proximo(obtener(e.computadoras, c).cola)) = Siguiente(Π_2(proximo(obtener(e.computadoras, c).cola))).prioridad $\wedge$
desencolar(obtener(e.computadoras, c).cola) = armarCola(obtener(e.computadoras, c).paquetes - {Siguiente(Π_2(proximo(obtener(e.computadoras, c).cola)))} $\wedge$
($\forall$ p: paquete, $p \in$ obtener(e.computadoras, c).paquetes) (
\\ *12* origen(p).ip $\in$ claves (e.computadoras) $\wedge$ destino(p).ip $\in$ claves (e.computadoras) $\wedge$
\\ *13* origen(p).ip $\neq$ destino(p).ip $\wedge$
\\ *14* hayCamino?(e.red, origen(p), destino(p)) $\wedge$
\\ *15* esta? (c, caminos[obtener(e.computadoras, origen(p).ip)][obtener(e.computadoras, destino(p).ip)) $\wedge$
\\ *16* ($\forall$ c': hostname, $c' \in$ claves(e.computadoras), $c' \neq c$) $\neg$($\exists$ p': paquete, $p' \in$ obtener(e.computadoras, c').paquetes, $p \neq p'$) p.id = p'.id
\\ *17* definido?(obtener(e.computadoras, c).paquetesPorID, p.id) $\wedge_L$
Siguiente(obtener(obtener(e.computadoras, c).paquetesPorID, p.id)) = p $\wedge$
\\ *18* ($\exists$ it: itConj(paquete), $it \in$ obtener(e.computadoras, c).cola) Siguiente(it) = p))

Especificación de las funciones auxiliares utilizadas en Rep

armarCola : conj(paquete) $\longrightarrow$ cola(paquete)

axiomas $\forall$ cc: conj(paquete)

armarCola(cc) $\equiv$ **if** $\emptyset$?(cc) **then**
Vacia()
else
encolar(dameUno(cc).prioridad, dameUno(cc), armarCola(sinUno(cc)))
fi

Abs : estr_dcnet e $\longrightarrow$ dcnet {Rep(e)}

Abs(e) $\equiv$ d | red(d) = e.red $\wedge$
(∀ c: compu, $c \in$ computadoras(red(d))) (
cantidadEnviados(d, c) = obtener(e.computadoras, c.ip).cantEnvios $\wedge$
enEspera(d, c) = obtener(e.computadoras, c.ip).paquetes $\wedge$
(∀ p: paquete, $p \in$ obtener(e.computadoras, c.ip).paquetes) caminoRecorrido (d, p) = e.caminos[obtener(e.computadoras, origen(p).ip).indice][obtener(e.computadoras, c.ip).indice])

Algoritmos

Algorithm 15 Implementación de Red

```
function iRed(in d: estr_dcnet)→ res: Red
    res ← d.red
end function
```

Algorithm 16 Implementación de CaminoRecorrido

```
function iCaminoRecorrido(in d: estr_dcnet, in p: IDPaquete)→ res: lista(hostname)
    itCompu ← CrearIt(d.computadoras)                                          ▷ O(1)
    yaEncontrado ← FALSE                                                       ▷ O(1)
    while HaySiguiente(itCompu) && ¬yaEncontrado do ▷ Guarda: O(1) ▷ Se repite a lo sumo n veces ▷ O(n)
        if Definido?(SiguienteSignificado(itCompu).paqPorID, p) then           ▷ O(log(k))
            paquete ← Significado(SiguienteSignificado(itCompu).paqPorID, p)   ▷ O(1)
            yaEncontrado ← TRUE                                                ▷ O(1)
        else
            Avanzar(itCompu)                                                   ▷ O(1)
        end if
    end while
    res ← caminos[Significado(d.computadoras, π3(paquete)).indice][SiguienteSignificado(itCompu).indice]
▷ O(1) + O(n) + O(1)
end function
```

Algorithm 17 Implementación de paquetes enviados

```
function iCantidadEnviados(in d: estr_dcnet, in c: hostname)→ res: nat
    it ← Significado(d.porHostname, c)                                         ▷ O(L)
    res ← SiguienteSignificado(it).cantEnvios                                  ▷ O(1)
end function
```

Algorithm 18 Implementación EnEspera

```
function iEnEspera(in d: estr_dcnet, in c: hostname)→ res : estr
    it ← Significado(d.porHostname, c)                                  ▷ O(L)
    res ← SiguienteSignificado(it).paquetes                             ▷ O(1)
end function
```

Algorithm 19 Implementación de iniciarDCNet

```
function iIniciarDCNet(in r : red)→ res: estr_dcnet
\\ creo un diccionario lineal
    diccCompus ← Vacio()                                                ▷ O(1)
\\ creo un diccionario universal(trie)
    diccHostname ← Vacio()                                              ▷ O(1)
    index ← 0                                                           ▷ O(1)
    itHostname ← CrearIt(Computadoras(r))                               ▷ O(1)
    masEnvios ← Siguiente(itHostname)                                   ▷ O(1)
\\ #Computadoras(r)) ó O(n)
    while HaySiguiente(itHostname) do                                   ▷ O(n) + O(1)
\\ no me queda clara la complejidad
        X ← <index, Vacio(), Vacio(), Vacio(), 0>                       ▷ O(1) + O(1) + O(1) + O(1) + O(1)
\\ ver complejidad
        itX ← DefinirRapido(diccCompus, Siguiente(itHostname), X)       ▷ O(copy(hostname)) +
O(copy(X))
\\ ver complejidad
        Definir(diccHostname, Siguiente(itHostname), itX)               ▷ O(L) + O(copy(X))
        index ← index + 1                                               ▷ O(1)
        Avanzar(itHostname)                                             ▷ O(1)
    end while
    itPC ← CrearIt(diccCompus)                                          ▷ O(1)
    itPC2 ← CrearIt(diccCompus)                                         ▷ O(1)
    n ← #Claves(diccCompus)                                             ▷ O(1)
    arrayCaminos ← CrearArreglo(n)                                      ▷ O(n)
\\ voy a crear un arreglo en cada posicion de arrayCaminos, el cual va a tener el minimo camino
\\ #Computadoras(r)) ó O(n)
    while HaySiguiente(itPC) do                                         ▷ O(#Computadoras(r))
        arrayDestinos ← CrearArreglo(n)                                 ▷ O(n)
\\
\\ #Computadoras(r)) ó O(n)
        while HaySiguiente(itPC2) do                                    ▷ O(#Computadoras(r))
            ConjCaminos ← CaminosMinimos(r, SiguienteClave(itPC), SiguienteClave(itPC2))  ▷ O(n*L)
            itConj ← CrearIt(ConjCaminos)                               ▷ O(1)
\\ de todos los caminos minimos me quedo con uno
            if HaySiguiente(itConj) then                                ▷ O(1)
                arrayDestinos[SiguienteSignificado(itPC2).indice] ← Siguiente(itConj)   ▷ O(1)
            else
\\ si no hay camino, creo una lista vacia
                arrayDestinos[SiguienteSignificado(itPC2).indice] ← Vacia()   ▷ O(1)
            end if
            Avanzar(itPC2)                                              ▷ O(1)
        end while
        arrayCaminos[SiguienteSignificado(itPC).indice] ← arrayDestinos ▷ O(1)
        Avanzar(itPC)                                                   ▷ O(1)
    end while
    res ← <Copiar(r), diccCompus, diccHostname, masEnvios, arrayCaminos >▷ O(Copiar(r)) + O(1) +
O(1) + O(1) + O(1)
end function
```

Algorithm 20 Implementación de crearPaquete

function ICREARPAQUETE(**in**/**out** d: `estr_dcnet`, **in** p: `paquete`)
itPC ← Significado(d.porHostname, paquete.origen) ▷ O(L)
\\ *el paquete se agregar por copia, esto hace que la complejidad pedida no se cumpla??*
itPaq ← AgregarRapido(SiguienteSignificado(itPC).paquetes, p) ▷ O(copy(p))
Encolar(SiguienteSignificado(itPC).cola, p.prioridad ,itPaq) ▷ O(log(n)), n cantidad de nodos)O(log(k)
Definir(SiguienteSignificado(itPC).paquetesPorID, IDpaquete, itPaq) ▷ O(log(k))
end function

Algorithm 21 Implementación de AvanzarSegundo

```
function IAvanzarSegundo(inout d: estr_dcnet)
    arreglo ← crearArreglo[#Claves(d.computadoras)] de tupla(usado: bool, paquete: paquete, destino: string), donde paquete es tupla(IDpaquete: nat, prioridad: nat, origen: string, destino: string)
                                                    ▷ O(n) para calcular cantidad de claves, O(1) para crearlo
    for (int i=0, < #Claves(d.computadoras), i++) do                    ▷ el ciclo se hará n veces
        arreglo[i].usado = false                                        ▷ O(1)
    end for

\\ Inicializo Iterador
    itCompu ← crearIt(d.computadoras)                                   ▷ O(1)
    i ← 0
                                                    ▷ Ciclo 1: Desencolo y guardo en arreglo auxiliar.
    while (HaySiguiente(itCompu)) do                                    ▷ el ciclo se hará a lo sumo n veces
        if (¬(Vacia?(SiguienteSignificado(itCompu).cola))) then         ▷ O(1)
\\ Borro el de mayor priorirdad del heap:
            itPaquete ← Desencolar(SiguienteSignificado(itCompu).cola)  ▷ O(log k)
\\ Lo elimino del dicc AVL
            Borrar(SiguienteSignificado(itCompu).paquetesPorID, Siguiente(itPaquete).IDpaquete)
                                                                        ▷ O(log k)
\\ Guardo el paquete en una variable
            paqueteDesencolado ← Siguiente(itPaquete)                   ▷ O(1)
\\ Lo elimino del conjunto lineal de paquetes
            EliminarSiguiente(itPaquete)                                ▷ O(1)
\\ Calculo proximo destino fijandome en la matriz
\\ El origen lo tengo en O(1) en el significado del iterador de compus.
            origen ← (SiguienteSignificado(itCompu)).indice             ▷ O(1)
\\ El destino lo obtengo en O(L) buscando por hostname el destino del paquete, y luego guardo el indice.
            itdestino ← Significado(d.porHostname, paqueteDesencolado.destino)  ▷ O(L)
            destino ← (SiguienteSignificado(itdestino)).indice          ▷ O(1)
            proxDest ← d.caminos[origen][destino][1]                    ▷ O(1)
\\ Lo inserto en el arreglo junto con el destino sólo si el destino no era el final.
            if (proxDest ≠ paqueteDesencolado.destino) then
                arreglo[i] ← <true, paqueteDesencolado, proxDest>       ▷ O(1)
            end if
\\ Aumento cantidad de envíos
            SiguienteSignificado(itCompu).cantEnvios ++                 ▷ O(1)
\\ Actualizo conMasEnvios
            envios ← SiguienteSignificado(itCompu).cantEnvios           ▷ O(1)
            if (envios > SiguienteSIgnificado(d.conMasEnvios).cantEnvios) then  ▷ O(1)
                d.conMasEnvios ← itCompu
            end if
        end if
\\ Avanzo de computadora
        Avanzar(itCompu)                                                ▷ O(1)
        i++
    end while
```

```
                                                ▷ Ciclo 2: Encolo los paquetes del vector a sus destinos correspondientes.
    i ← 0
    while HaySiguiente(itCompu) do                          ▷ el ciclo se hará a lo sumo n veces
        if arreglo[i].usado then
\\ Busco el proxDestino guardado en el arreglo por hostname.
            itdestino ← Significado(d.porHostname, arreglo[i].destino)                    ▷ O(L)
\\ Agrego el paquete al conjunto de paquetes del prox destino.
            itpaquete ← AgregarRapido(SiguienteSignificado(itdestino).paquetes, arreglo[i].paquete)
                                                                                          ▷ O(1)
\\ Encolo el heap del destino
            prioridad ← (arreglo[i].paquete).prioridad
            Encolar(SiguienteSignificado(itdestino).cola, prioridad, itpaquete)           ▷ O(log k)
\\ Lo agrego en el dicc AVL.
            IDpaq ← (arreglo[i].paquete).IDpaquete                                        ▷ O(1)
            Definir(SiguienteSignificado(itdestino).paquetesPorID, IDpaq, itpaquete)      ▷ O(log k)
        end if
        i++
        Avanzar(itCompu)
    end while
end function
```

Algorithm 22 Implementación de PaqueteEnTransito?

```
function iPaqueteEnTransito?(in d: estr_dcnet, in p:IDpaquete)→ res: bool
    res ← false                                                                           ▷ O(1)
    itCompu ← crearIt(d.computadoras)                                                     ▷ O(1)
    while HaySiguiente(itCompu) && ¬res do
▷ a lo sumo n veces, la guarda es O(1)
        itPaq ← crearIt(siguienteSignificado(itCompu).paquetes)                           ▷ O(1)
        while (HaySiguiente(itPaq) && Siguiente(itPaq).id ≠ p) do      ▷ a lo sumo k veces, la guarda es
O(1)
            Avanzar(itPaq)                                                                ▷ O(1)
        end while
        if Siguiente(itPaq) == p) then                                                    ▷ O(1)
            res ← True                                                                    ▷ O(1)
        end if
        Avanzar(itCompu)                                                                  ▷ O(1)
    end while
end function
```

Algorithm 23 Implementación de LaQueMasEnvió

```
function iLaQueMasEnvió(in d: estr_dcnet)→ res: hostname
    res ← SiguienteClave(d.conMasEnvios)
end function
```

Algorithm 24 Implementación de ==

```
function IIGUALDAD(in d1: estr_dcnet, in d2: estr_dcnet)→ res: bool
\\ Comparo redes usando == de red
    res ← (d1.red == d2.red)                                              ▷ O(???)
    if (res) then                                                         ▷ O(1)
        itCompu ← crearIt(d1.computadoras)                                ▷ O(1)
        string host                                                       ▷ O(1)
\\ Recorro las computadoras
        while (HaySiguiente(itCompu) && res) do   ▷ itero O(n) veces, la guarda es O(1)
            host ← SiguienteClave(itCompu)                                ▷ O(1)
\\ Comparo enEspera usando == de conjunto lineal, y cant. enviados
            res ← (enEspera(d1, host) == enEspera(d2, host) &&
cantidadEnviados(d1,host) == cantidadEnviados(d2,host))                   ▷ O(???)
            itpaq ← crearIt(SiguienteSignificado(itCompu).paquetes)       ▷ O(1)
            int j ← 0                                                     ▷ O(1)
            nat id                                                        ▷ O(1)
\\ Recorro paquetes de cada computadora
            while (HaySiguiente(itpaq) && res ) do   ▷ itero O(k) veces, la guarda es O(1)
                id ← Siguiente(itpaq).IDpaquete                           ▷ O(1)
\\ Comparo caminosRecorridos usando == de listas enlazadas
                res ← (caminoRecorrido(d1, id) == caminoRecorrido(d2, id))  ▷ O(???)
                avanzar(itpaq)                                            ▷ O(1)
            end while
            avanzar (itCompu)                                             ▷ O(1)
        end while
    end if
end function
```

3. Módulo Cola de Prioridad Logaritmica (α)

Interfaz

usa: TUPLA, NAT, BOOL, α.

se explica con: COLA DE PRIORIDAD ALTERNATIVA.

géneros: `colaLog(`α`)`.

Operaciones de Cola de Prioridad *HEAP*

VACIA() $\rightarrow$ *res* : `colaLog(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ vacia}
Complejidad: $O(1)$
Descripción: Crea una cola vacia.

VACIA?(**in** *estr*: `colaLog(`α`)`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ vacia?($estr$)}
Complejidad: $O(1)$
Descripción: Indica si la cola esta vacia.

PRÓXIMO(**in** *estr*: `colaLog(`α`)`) $\rightarrow$ *res* : `tupla(`nat, α`)`
Pre $\equiv$ {$\neg Vacia?(estr)$}
Post $\equiv$ {$res =_{obs} proximo(estr)$}
Complejidad: $O(copiar(\alpha))$
Descripción: Devuelve una tupla que contiene al próximo elemento y su prioridad.

ENCOLAR(**in/out** *estr*: `colaLog(`α`)`, **in** *prio*: `nat`, **in** *valor*: α) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {estr = estr_0}
Post $\equiv$ {$res \wedge$ estr $=_{obs}$ encolar(estr_0)}
Complejidad: $O(log(n) + copiar(\alpha))$
Descripción: Crea un nuevo elemento con los parametros dados y lo agrega a la cola.

DESENCOLAR(**in/out** *estr*: `colaLog(`α`)`) $\rightarrow$ *res* : α
Pre $\equiv$ {estr = $estr_0 \wedge \neg Vacia?(estr)$}
Post $\equiv$ {estr $=_{obs}$ desencolar($estr_0$) $\wedge$ $res =_{obs} proximo(estr_0)$}
Complejidad: $O(log(n) + copiar(\alpha) + borrar(\alpha))$
Descripción: Devuelve al elemento de mayor prioridad y lo remueve de la cola. La cola no debe estar vacía.

TAD COLA DE PRIORIDAD ALTERNATIVA(α)

géneros colaPrio(α)

exporta colaPrio(α), generadores, observadores

usa BOOL, NAT, TUPLA

observadores básicos

vacía?	: colaPrior(α)	$\longrightarrow$	bool	
próximo	: colaPrior(α) c	$\longrightarrow$	tupla(nat, α)	$\{\neg$ vacía?(c)$\}$
desencolar	: colaPrior(α) c	$\longrightarrow$	colaPrior(α)	$\{\neg$ vacía?(c)$\}$

generadores

vacía	:	$\longrightarrow$	colaPrior(α)
encolar	: nat $\times$ α $\times$ colaPrior(α)	$\longrightarrow$	colaPrior(α)

axiomas $\forall$ c: colaPrior(α), $\forall$ e: α

vacía?(vacía)	$\equiv$	true
vacía?(encolar(p, e, c))	$\equiv$	false
próximo(encolar(p, e, c))	$\equiv$	**if** vacía?(c) $\vee_{\text{L}}$ $\Pi_1(proximo(c)) < p$ **then** $< p, e >$ **else if** $\Pi_1(proximo(c)) = p$ **then** $< p, e > \vee\ proximo(c)$ **else** $proximo(c)$ **fi fi**
desencolar(encolar(p, e, c))	$\equiv$	**if** vacía?(c) $\vee_{\text{L}}$ $\Pi_1(proximo(c)) < p$ **then** c **else if** $\Pi_1(proximo(c)) = p$ **then** $c \vee encolar(p, e, desencolar(c)$ **else** $encolar(p, e, desencolar(c)$ **fi fi**

Fin TAD

Representación

colaLog(α) se representa con estr_heap(α)

donde estr_heap(α) es tupla(*size*: nat
primero: nodo(α)
)

donde nodo(α) es tupla(*padre*: puntero(nodo(α))
izq: puntero(nodo(α))
der: puntero(nodo(α))
prio: nat
valor: α
)

Rep: estr_heap(α) $\rightarrow$ bool
Rep($estr$) $\equiv$ true $\iff$ $size = \#arbol(estr.primero) \ \wedge_L$
$((estr.primero).padre = null \wedge$
$(\forall e : estr\_heap)(e \in arbol(estr.primero) \wedge \ e \neq (estr.primero) \Rightarrow (e.padre \neq null \wedge_L (((e.padre).izq = e \vee (e.padre).der = e) \wedge \ \neg(((e.padre).izq = e \wedge (e.padre).der = e))))) \wedge$
$(\forall e : estr\_heap)(e \in arbol(estr.primero) \Rightarrow ((estr.izq \neq null \Rightarrow estr.prio \geq (estr.izq).prio) \wedge (estr.der \neq null \Rightarrow estr.prio \geq (estr.der).prio))) \wedge$
$(\forall e : estr\_heap)(e \in arbol(estr.primero) \Rightarrow caminoHastaRaiz(e, arbol(estr.primero)) \leq \lfloor log_2(size) \rfloor + 1))$

Abs: estr_heap(α) $e \rightarrow$ colaPrio(α) { Rep(e) }
Abs(e) $\equiv$ c: colaPrio(α) | $Vacia?(e) = Vacia?(c) \ \wedge_L$
$(\neg Vacia?(e) \Rightarrow Proximo(e) = Proximo(c)) \ \wedge$
$(\neg Vacia?(e) \Rightarrow Desencolar(e) = Desencolar(c))$

Especificación de las operaciones auxiliares utilizadas para Rep y Abs

arbol : nodo(α) $\longrightarrow$ conj(nodo(α))

caminoHastaRaiz : nodo(α) $\longrightarrow$ nat

arbol(n) $\equiv$ **if** $n.izq \neq null \wedge n.der \neq null$ **then**
Ag($n.valor, arbol(n.izq) \cup arbol(n.der)$)
else
if $n.izq \neq null$ **then**
Ag($n.valor, arbol(n.izq)$)
else
if $n.der \neq null$ **then** Ag($n.valor, arbol(n.der)$) **else** Ag($n.valor, \emptyset$) **fi**
fi
fi

caminoHastaRaiz(n) $\equiv$ **if** $n.padre = null$ **then** 0 **else** $caminoHastaRaiz(n.padre) + 1$ **fi**

Algoritmos

Algorithm 25 Implementación de Vacia

```
function iVacia→ res: colaLog(α)
    res ← <0, null>                                          ▷ O(1)
end function
```

Algorithm 26 Implementación de Vacia?

```
function iVacia?(in estr : estr_heap(α))→ res: bool
    res ← (estr.primero == null)                             ▷ O(1)
end function
```

Algorithm 27 Implementación de Próximo

```
function Próximo(in estr : estr_heap(α))→ res: tupla(nat, α)
    res ← < (estr.primero).prioridad, (estr.primero).valor >  ▷ O(copiar(α))
end function
```

Algorithm 28 Implementación de Encolar

```
function iEncolar(in/out estr: estr_heap(α), in prio: nat, in valor: α)→ res: bool
    res ← true                                                                  ▷ O(1)
    if estr.size == 0 then                                                      ▷ O(1)
        estr.primero ← puntero(<null, null, null, prio, valor>)                 ▷ O(copiar(α))
    else
        size++                                                                  ▷ O(1)
        x ← sizer                                                               ▷ O(1)
        y ← <>                                                                  ▷ O(1)
        while x ≠ 0 do ▷ La cantidad de veces que se ejecuta el ciclo es igual a la altura del heap. Al ser un arbol binario completo, la altura siempre será O(log(n))
            y ← (x%2) • y                                                       ▷ O(1)
            x ← x/2                                                             ▷ O(1)
        end while
        y ← com(y)                                                              ▷ O(log(n))
        z ← estr.primero                                                        ▷ O(1)
        y ← fin(y)                                                              ▷ O(1)
        while long(y)> 1 do                                                     ▷ El ciclo se ejecuta O(log(n)) veces
            z ← if prim(y) == 0 then z.izq else z.der                           ▷ O(1)
            y ← fin(y)                                                          ▷ O(1)
        end while
        w ← <null, null, null, prio, valor>                                     ▷ O(copiar(α))
        w.padre ← z                                                             ▷ O(1)
        if prim(y) == 0 then                                                    ▷ O(1)
            z.izq ← w                                                           ▷ O(1)
        else
            z.der ← w                                                           ▷ O(1)
        end if
        while w ≠ estr.primero ∧_L w.prio > (w.padre).prio do ▷ La cantidad de veces que se ejecuta el ciclo es a lo sumo la altura del heap, que es O(log(n))
            aux ← w.valor                                                       ▷ O(copiar(α))
            w.valor ← (w.padre).valor                                           ▷ O(copiar(α))
            (w.padre).valor ← aux                                               ▷ O(copiar(α))
            aux2 ← w.prio                                                       ▷ O(1)
            w.prio ← (w.padre).prio                                             ▷ O(1)
            (w.padre).prio ← aux2                                               ▷ O(1)
            w ← w.padre                                                         ▷ O(1)
        end while
    end if
end function
```

Algorithm 29 Implementación de Desencolar

function IDESENCOLAR(**in/out** *estr*: `estr_heap(`α`)`)$\rightarrow$ *res*: α
res $\leftarrow$ $*$(*estr.primero*) ▷ O(1)
$x \leftarrow$ *size* ▷ O(1)
$y \leftarrow$ $<>$ ▷ O(1)
while x $\neq$ 0 **do** ▷ La cantidad de veces que se ejecuta el ciclo es igual a la altura del heap. Al ser un arbol binario completo, la altura siempre será O(log(n))
$y \leftarrow (x\,\%2) \bullet y$ ▷ O(1)
$x \leftarrow x/2$ ▷ O(1)
end while
$y \leftarrow$ com(y) ▷ O(log(n))
$z \leftarrow$ *estr.primero* ▷ O(1)
$y \leftarrow$ fin(y) ▷ O(1)
while long$(y)>1$ **do** ▷ El ciclo se ejecuta O(log(n)) veces
$z \leftarrow$ **if** prim$(y) == 0$ **then** *z.izq* **else** *z.der* ▷ O(1)
$y \leftarrow$ fin(y) ▷ O(1)
end while
$w \leftarrow <null, null, null, prio, valor>$ ▷ O(copiar(α))
$w.padre \leftarrow z$ ▷ O(1)
if prim$(y) == 0$ **then** ▷ O(1)
$z.izq \leftarrow$ *puntero*(w) ▷ O(1)
else
$z.der \leftarrow$ *puntero*(w) ▷ O(1)
end if
$(estr.primero).valor \leftarrow$ *z.valor* ▷ O(copiar(α))
$(estr.primero).prio \leftarrow$ *z.prio* ▷ O(1)
borrar(z) ▷ O(borrar(α))
$z \leftarrow$ *estr.primero* ▷ O(1)
size $--$ ▷ O(1)
while $(z.izq \neq null \ \vee z.der \neq null) \ \wedge_L \ z.prio <$ *maxPrio*$(z.izq, z.der)$ **do** ▷ La cantidad de veces que se ejecuta el ciclo es a lo sumo la altura del heap, que es O(log(n))

\\ maxPrio devuelve la maxima prioridad si ambos punteros son validos, o la prioridad apuntada por el puntero no nulo en caso de que alguno no lo sea
if $z.der == null \ \vee_L \ (z.izq).prio \geq (z.der).prio$ **then** ▷ O(1)
aux $\leftarrow$ *z.valor* ▷ O(copiar(α))
$z.valor \leftarrow (z.izq).valor$ ▷ O(copiar(α))
$(z.izq).valor \leftarrow$ *aux* ▷ O(copiar(α))
$aux2 \leftarrow$ *z.prio* ▷ O(1)
$z.prio \leftarrow (z.izq).prio$ ▷ O(1)
$(z.izq).prio \leftarrow aux2$ ▷ O(1)
$z \leftarrow z.izq$ ▷ O(1)
else
aux $\leftarrow$ *z.valor* ▷ O(copiar(α))
$z.valor \leftarrow (z.der).valor$ ▷ O(copiar(α))
$(z.der).valor \leftarrow$ *aux* ▷ O(copiar(α))
$aux2 \leftarrow$ *z.prio* ▷ O(1)
$z.prio \leftarrow (z.der).prio$ ▷ O(1)
$(z.der).prio \leftarrow aux2$ ▷ O(1)
$z \leftarrow z.der$ ▷ O(1)
end if
end while
end function

4. Módulo Diccionario Universal (σ)

Interfaz

se explica con: DICCIONARIO(STRING, σ).

géneros: `diccUniv(`κ, σ`)`.

Operaciones

DEFINIDA(**in** d: `diccUniv(STRING, `σ`)`, **in** c: `STRING`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ def?(c,d)}
Complejidad: O(L), donde L es la cantidad de caracteres de la clave más grande.
Descripción: Indica si la clave dada está definida en el diccionario.

OBTENER(**in** d: `diccUniv(STRING, `σ`)`, **in** c: `STRING`) $\rightarrow$ res : σ
Pre $\equiv$ {def?(c,d)}
Post $\equiv$ {res $=_{\text{obs}}$ obtener(c,d)}
Complejidad: O(L)
Descripción: Devuelve el significado asociado a la clave dada.
Aliasing: Devuelve al significado por alias.

VACIO() $\rightarrow$ res : `diccUniv(STRING,`σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ vacio()}
Complejidad: O(1)
Descripción: Crea un diccionario vacío.

DEFINIR(**in/out** d: `diccUniv(STRING,`σ`)`, **in** c: STRING, **in** s: σ) $\rightarrow$ res : `Bool`
Pre $\equiv$ {$\neg$def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {$d = definir(c, d_0)$}
Complejidad: $O(L) + O(copiar(\sigma)$
Descripción: Agrega la clave al diccionario, asociándole el significado dado como parámetro. res indica si la clave ya estaba definida.

BORRAR(**in/out** d: `diccUniv(STRING,`σ`)`, **in** c: `STRING`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {d=d_0}
Post $\equiv$ {d=borrar(c,d_0)}
Complejidad: $O(L) + O(borrar(\sigma)$
Descripción: Borra la clave dada y su significado del diccionario. res indica si la clave estaba definida (su valor es $true$ en caso de estarlo).

CLAVES(**in** d: `diccUniv(STRING,`σ`)`) $\rightarrow$ res : `conj(STRING)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ claves(d)}
Complejidad: O(n*L), donde n es la cantidad de claves.
Descripción: Devuelve el conjunto de las claves del diccionario.

\\ *Diseño provisto por la cátedra.*

5. Módulo Diccionario Logaritmico (κ, σ)

Interfaz

usa: BOOL, NAT.

se explica con: DICCIONARIO(κ, σ).

géneros: `diccLog(`κ, σ`)`.

Operaciones

DEFINIDO?(**in** d: `diccLog(`κ, σ`)`, **in** c: κ) $\rightarrow$ res : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ def?(c,d)}
Complejidad: $O(log(n) * comparar(\kappa))$

SIGNIFICADO(**in** d: `diccLog(`κ, σ`)`, **in** c: κ) $\rightarrow$ res : σ
Pre $\equiv$ {def?(c,d)}
Post $\equiv$ {res $=_{\text{obs}}$ obtener(c,d)}
Complejidad: $O(log(n) * comparar(\kappa) + copiar(\sigma)$

VACIO() $\rightarrow$ res : `diccLog(`κ, σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ vacio()}
Complejidad: $O(1)$

DEFINIR(**in/out** d: `diccLog(`κ, σ`)`, **in** c: κ, **in** s: σ)
Pre $\equiv$ {$\neg$def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d=definir(c,d_0)}
Complejidad: O(log(n))

BORRAR(**in/out** d: `diccLog(`κ, σ`)`, **in** c: κ)
Pre $\equiv$ {def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d=borrar(c,d_0)}
Complejidad: $O(log(n) * comparar(\kappa) + max(borrar(\kappa), borrar(\sigma)) + max(copiar(\kappa), copiar(\sigma)))$

Representación

`diccLog` se representa con `puntero(estr(`κ, σ`)`

donde `estr(`κ, σ`)` es `tupla(`*clave*: κ
izquierdo: `puntero(nodo(`κ, σ`))`
derecho: `puntero(nodo(`κ, σ`))`
significado: σ
nivel: `nat`)

\\ *http://user.it.uu.se/~arnea/abs/simp.html*

Rep: estr $\rightarrow$ bool

$\text{Rep}(e) \equiv \text{true} \iff ((\forall n_1, n_2 : estr(\kappa, \sigma))(n_1 \in arbol(e) \wedge n_2 \in arbol(e) \Rightarrow_L$
$(n_1.clave < n_2.clave \Rightarrow n_1 \in arbol(n_2.izquierdo)) \wedge (n_1.clave > n_2.clave \Rightarrow n_1 \in arbol(n_2.derecho)) \wedge$
$((\forall n_1, n_2 : estr(\kappa, \sigma))(n_1 \in arbol(e) \wedge n_2 \in arbol(e) \wedge n_1.clave \neq n_2.clave \Rightarrow_L$
$(n_1.izquierdo = n_2.izquierdo \vee n_1.izquierdo = n_2.derecho \Rightarrow n_1.izquierdo = NULO) \wedge$
$(n_1.derecho = n_2.izquierdo \vee n_1.derecho = n_2.derecho \Rightarrow n_1.derecho = NULO)) \wedge$
$((\forall n : estr(\kappa, \sigma))(n \in arbol(e) \wedge n.izquierdo = NULO \wedge n.derecho = NULO \Rightarrow n.nivel = 1) \wedge$
$((\forall n : estr(\kappa, \sigma))(n \in arbol(e) \wedge n.izquierdo \neq NULO \Rightarrow (n.izquierdo).nivel = n.nivel - 1) \wedge$
$((\forall n : estr(\kappa, \sigma))(n \in arbol(e) \wedge n.derecho \neq NULO \Rightarrow$
$((n.derecho).nivel = n.nivel - 1 \vee (n.derecho).nivel = n.nivel)) \wedge$
$((\forall n : estr(\kappa, \sigma))(n \in arbol(e) \wedge n.derecho \neq NULO \wedge_L (n.derecho).derecho \neq NULO \Rightarrow_L$
$((n.derecho).derecho).nivel < n.nivel) \wedge$
$((\forall n : estr(\kappa, \sigma))(n \in arbol(e) \wedge n.nivel > 1 \Rightarrow (n.izquierdo \neq NULO \wedge n.derecho \neq NULO)$

Abs: $diccLog(\kappa, \sigma)\ d \rightarrow dicc(\kappa, \sigma)$ {Rep(d) }

$\text{Abs}(d) \equiv$ c: dicc(κ, σ) | $((\forall k : \kappa)(k \in claves(c) \Rightarrow (\exists n : estr(\kappa, \sigma))(n \in arbol(d) \wedge n.clave = k)) \wedge$
$((\forall n : estr(\kappa, \sigma))(n \in arbol(d) \Rightarrow n.clave \in claves(c)))) \wedge_L$
$(\forall n : estr(\kappa, \sigma))(n \in arbol(d) \Rightarrow obtener(c, n.clave) =_{obs} n.significado)$

Especificación de las operaciones auxiliares utilizadas para Rep y Abs
arbol: $puntero(estr(\kappa\,\sigma)) \rightarrow conj(puntero(\kappa, \sigma))$

arbol(n) $\equiv$ **if** $n.izq \neq null \wedge n.der \neq null$ **then**
Ag(&$n, arbol(n.izq) \cup arbol(n.der)$)
else
if $n.izq \neq null$ **then**
Ag(&$n, arbol(n.izq)$)
else
if $n.der \neq null$ **then** Ag(&$n, arbol(n.der)$) **else** Ag(&$n, \emptyset$) **fi**
fi
fi

Algoritmos

Algorithm 30 Implementación de Definido?

```
function iDefinido?(in d: estr , in c: κ)→ res: bool
    nodoActual ← d                                                          ▷ O(1)
    res ← FALSE                                                             ▷ O(1)
    while ¬(nodoActual == NULO) && ¬res do   ▷ El ciclo se ejecuta en el peor caso una cantidad de
veces igual a la altura del arbol. Al ser auto-balanceado, su altura siempre sera O(log(n))
        if nodoActual.clave == c then                                       ▷ O(comparar(κ))
            res ← TRUE                                                      ▷ O(1)
        else
            if c <nodoActual.clave then                                     ▷ O(comparar(κ))
                nodoActual ← nodoActual.izquierdo                           ▷ O(1)
            else
                nodoActual ← nodoActual.derecho                             ▷ O(1)
            end if
        end if
    end while
end function
```

Algorithm 31 Implementación de Significado

```
function iSignificado(in d: estr , in c: κ)→ res: σ
    nodoActual ← d                                                          ▷ O(1)
    while ¬(nodoActual == NULO) && ¬res do   ▷ El ciclo se ejecuta en el peor caso O(log(n)) veces.
        if nodoActual.clave == c then                                       ▷ O(comparar(κ))
            res ← nodoActual.significado   ▷ O(copiar(σ)). Esta operacion solo se ejecuta una vez (implica
¬guarda del ciclo que la contiene).
        else
            if c <nodoActual.clave then                                     ▷ O(comparar(κ))
                nodoActual ← nodoActual.izquierdo                           ▷ O(1)
            else
                nodoActual ← nodoActual.derecho                             ▷ O(1)
            end if
        end if
    end while
end function
```

Algorithm 32 Implementación de Vacio

```
function iVacio→ res: estr
    res ← NULO                                                              ▷ O(1)
end function
```

Algorithm 33 Implementación de Definir

```
function iDefinir(inout d: estr , in c: κ, in s: σ)
    if d == NULO then                                                          ▷ O(1)
        res ← < c, NULO, NULO, s, 1 >                                          ▷ O(max(copiar(κ), copiar(σ))
    else if c < d.clave then                                                   ▷ O(comparar(κ)
        d.izquierdo ← iDefinir(d.izquierdo, c, s)  ▷ En el peor caso se llama recursivamente a la funcion
una cantidad de veces igual a la altura del arbol, que es O(log(n)).
    else if c > d.clave then                                                   ▷ O(comparar(κ))
        d.derecho ← iDefinir(d.derecho, c, s) ▷ En el peor caso se llama recursivamente a la funcion una
cantidad de veces igual a la altura del arbol, que es O(log(n)).
    end if
    d ← Torsion(d)                                                             ▷ O(1)
    d ← Division(d)                                                            ▷ O(1)
end function
```

Algorithm 34 Implementación de Torsion

```
function iTorsion(in d: estr)→ res: estr
    if d == NULO || d.izquierdo == NULO then                                   ▷ O(1)
        res ← d                                                                ▷ O(1)
    else
        if (d.izquierdo).nivel ≥ d.nivel then                                  ▷ O(1)
            nodoAux ← d.izquierdo                                              ▷ O(1)
            d.izquierdo ← nodoAux.derecho                                      ▷ O(1)
            nodo.derecho ← d                                                   ▷ O(1)
            res ← nodoAux                                                      ▷ O(1)
        else
            res ← d                                                            ▷ O(1)
        end if
    end if
end function
```

Algorithm 35 Implementación de Division

```
function iDivision(in d: estr)→ res: estr
    if d == NULO || d.derecho == NULO || d.derecho.derecho == NULO then        ▷ O(1)
        res ← d                                                                ▷ O(1)
    else
        if (d.derecho.derecho).nivel == d.nivel then                           ▷ O(1)
            nodoAux ← d.derecho                                                ▷ O(1)
            d.derecho ← nodoAux.izquierdo                                      ▷ O(1)
            nodoAux.izquierdo ← d                                              ▷ O(1)
            nodoAux.nivel++                                                    ▷ O(1)
            res ← nodoAux                                                      ▷ O(1)
        else
            res ← d                                                            ▷ O(1)
        end if
    end if
end function
```

Algorithm 36 Implementación de Borrar

```
function iBorrar(inout d: estr, in c: κ)
    if d == NULO then                                                          ▷ O(1)
        endFunction
    else if c > d.clave then                                                   ▷ O(comparar(κ))
        d.derecho ← iBorrar(d.derecho, c)   ▷ En el peor caso se llama recursivamente a la funcion una
cantidad de veces igual a la altura del arbol, que es O(log(n)).
    else if c < d.clave then                                                   ▷ O(comparar(κ))
        d.izquierdo ← iBorrar(d.izquierdo, c) ▷ En el peor caso se llama recursivamente a la funcion una
cantidad de veces igual a la altura del arbol, que es O(log(n)).
    else if d.izquierdo == NULO ∧ d.derecho == NULO then                       ▷ O(1)
        borrar(d)                                                              ▷ O(max(borrar(κ), borrar(σ)))
        d ← NULO                                                               ▷ O(1)
    else if d.izquierdo == NULO then                                           ▷ O(1)
        aux ← d.derecho                                                        ▷ O(1)
        while aux.izquierdo ≠ NULO do          ▷ En el peor caso el ciclo se ejecuta O(log(n)) veces.
            aux ← aux.izquierdo                                                ▷ O(1)
        end while
        d.derecho ← iBorrar(aux.clave, d.derecho)
        d.clave ← aux.clave                                                    ▷ O(copiar(κ))
        d.significado ← aux.significado                                        ▷ O(copiar(σ))
    else
        aux ← d.izquierdo                                                      ▷ O(1)
        while aux.derecho ≠ NULO do            ▷ En el peor caso el ciclo se ejecuta O(log(n)) veces.
            aux ← aux.derecho                                                  ▷ O(1)
        end while
        d.izquierdo ← iBorrar(aux.clave, d.izquierdo)
        d.clave ← aux.clave                                                    ▷ O(copiar(κ))
        d.significado ← aux.significado                                        ▷ O(copiar(σ))
    end if
    d ← Nivelar(T)                                                             ▷ O(1)
    d ← Torsion(T)                                                             ▷ O(1)
    if d.derecho ≠ NULO then                                                   ▷ O(1)
        (d.derecho).derecho ← Torsion((d.derecho).derecho)                     ▷ O(1)
    end if
    d ← Division(T)                                                            ▷ O(1)
    d.derecho ← Division(d.derecho)                                            ▷ O(1)
    res ← d                                                                    ▷ O(1)
end function
procedure Nivelar(inout d: estr)
    nivel_correcto ← min((d.izquierdo).nivel, (d.derecho).nivel) + 1           ▷ O(1)
    if nivel_correcto < d.nivel then                                           ▷ O(1)
        d.nivel ← nivel_correcto                                               ▷ O(1)
        if nivel_correcto < (d.derecho).nivel then                             ▷ O(1)
            (d.derecho)nivel ← nivel_correcto                                  ▷ O(1)
        end if
    end if
end procedure
```