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)}
Complejidad: O(n^6 + n^5 * L)
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.

• == •(**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: O(n^3)
Aliasing: copia la red.
Requiere: 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 × hostname s $\longrightarrow$ compu	$\{s \in$ hostnames(r)$\}$
auxDameCompu	: red r × hostname s × 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($secu$)) • 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`.

▷ Al no tener que cumplir con complejidades utilizamos un diccionario con los hostnames como claves. El significado de cada hostname corresponde a una tupla con datos de la computadora con ese hostname.

\\ 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) ▷ O(n^4 + n^3 x L)
    ActualizarCaminos(r, c2, c1) ▷ O(n^4 + n^3 x L)
\\ *Creo conjunto con los actualizados hasta el momento*
    actualizados ← Vacio() ▷ conjunto
    AgregarRapido(actualizados, c1) ▷ O(L)
    AgregarRapido(actualizados, c2) ▷ O(L)
\\ *Actualizo caminos del resto de la Red por recursion*
    ActualizarVecinos(r, c1, actualizados) ▷ O(n^6 + n^5 * L)
end function ▷ O(n^6 + n^5 * L)

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(n)
    while (HaySiguiente(itAlcanzables2) do        ▷ se ejecuta a lo sumo n-1 veces     ▷ O(n) x
\\ Recorro alcanzables de c1
        itAlcanzables1 ← crearIt(Significado(r, c1).alcanzables)                      ▷ O(n)
        while (HaySiguiente(itAlcanzables1)) do      ▷ se ejecuta a lo sumo n-1 veces  ▷ O(n) x
            if (SiguienteClave(itAlcanzables2) == SiguienteClave(itAlcanzables1)) then  ▷ 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(n)
                itCaminos ← crearIt (SiguienteSignificado(itAlcanzables1))             ▷ O(1)
                camino1 ← Siguiente(itCaminos)            ▷ camino minimo del c1       ▷ O(n)
                if (longitud(camino1) >longitud(camino2)) then      ▷ cada camino tiene a lo sumo n
elementos                                                                              ▷ O(1)
\\ Los caminos nuevos son mas cortos, borro los que estÃ¡n y copio los nuevos
                    Borrar(Significado(r, c1).alcanzables, SiguienteClave(itAlcanzables1))  ▷ O(n)
\\ 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                                ▷ O(n) x
                        nuevoCamino ← copy(Siguiente(itCaminos))  ▷ copio el camino que voy a modificar
▷ O(n)
                        AgregarAdelante(nuevoCamino, c1)                               ▷ O(L)
                        AgregarRapido (caminos, nuevoCamino)                           ▷ O(n)
                        Avanzar(itCaminos)                                             ▷ O(1)
                    end while                                                  ▷ O(n x (n + L))
\\ agrego el nuevo alcanzable con el camino
                    DefinirRapido(Significado(r,c1).alcanzables, SiguienteClave(itAlcanzables2), caminos) ▷
O(n) + O(L)
                else
                    if (longitud(camino1) == longitud(camino2)) then                   ▷ O(1)
\\  Tengo que agregar los nuevos caminos (modificados) al conjunto de caminos actual
                        itCaminos ← crearIt(SiguienteSignificado(itAlcanzables2))      ▷ O(1)
                        while (HaySiguiente(itCaminos)) do                             ▷ O(n) x
                            nuevoCamino ← copy(Siguiente(itCaminos))     ▷ copio el camino que voy a
modificar                                                                              ▷ O(n)
                            AgregarAdelante(nuevoCamino, c1)                           ▷ O(L)
                            Agregar(SiguienteSignificado(itAlcanzables1), nuevoCamino) ▷ O(n)
                            Avanzar(itCaminos)                                         ▷ O(1)
                        end while                                              ▷ O(n x (n + L))
                    end if
                end if
            else
\\ 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                                    ▷ O(n) x
                    nuevoCamino ← copy(Siguiente(itCaminos))  ▷ copio el camino que voy a modificar  ▷
O(n)
                    AgregarAdelante(nuevoCamino, c1)                                   ▷ O(L)
                    AgregarRapido (caminos, nuevoCamino)                               ▷ O(n)
                    Avanzar(itCaminos)                                                 ▷ O(1)
                end while                                                      ▷ O(n x (n + L))
                DefinirRapido(Significado(r, c1).alcanzables, SiguienteClave(itAlcanzables2), caminos)
            end if
```

```
        Avanzar(itAlcanzables1)                                        ▷ O(1)
      end while                                    ▷ O(n x (n + (n x (n + L))) )
      Avanzar(itAlcanzables2)                                          ▷ O(1)
    end whileO(n + n x (n x (n + (n x (n + L)))
  end function                                              ▷ O(n^4 + n^3 x L)
```

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)                 ▷ O(n)
    while ( HaySiguiente(itVecinos) ) do                               ▷ O(n x)
  \\ Si todavÃa no fue actualizado, lo actualizo y hago recursión sobre los vecinos.
      if (SiguienteClave(itVecinos) ∉ actualizados) then               ▷ O(n x L)
        ActualizarCaminos(r, SiguienteClave(itVecinos), c )   ▷ O(n^4 + n^3 x L)
        AgregarRapido (actualizados, SiguienteClave(itVecinos))         ▷ O(L)
        ActualizarVecinos(r, SiguienteClave(itVecinos), actualizados) ▷ recursión hasta actualizar las n
  computadoras, O(n) x
      end if
      Avanzar(itVecinos)                                                ▷ O(1)
    end while
  end function                                              ▷ O(n^6 + n^5 * L)
```

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(1)
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(n * log(k))
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(N*n*(L+n)
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(L + log(k))
Descripción: agrega un paquete a la red.
Aliasing: el paquete se agrega por copia.

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

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

LAQUEMASENVIO(**in** d: `dcnet`) → res : `hostname`
Pre ≡ {true}
Post ≡ $\{res =_{obs}$ laQueMasEnvio(d).ip$\}$
Complejidad: O(1)
Descripción: devuelve la computadora que más paquetes envió.
Aliasing: res se devuelve por copia.

• = •(**in** d_1 : `dcnet`, **in** d_2 : `dcnet`) → res : `bool`
Pre ≡ {true}
Post ≡ $\{res =_{obs} (d_1 =_{obs} d_2\}$
Complejidad: O(n*n * (L + n*n + m + log(k)) + n*(m*m + L))
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 × hostname s $\longrightarrow$ compu
{$s \in$ dameHostnames(computadoras(red(d)))}

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

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

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

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

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

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

dameHostnames(cc) ≡ **if** vacio?(cc) **then**
  ∅
else
  Ag(ip(dameUno(cc)), dameHostnames(sinUno(cc)))
fi

dameCompu(d, s) ≡ auxDameCompu(s, computadoras(red((d)))

auxDameCompu(s, cc) ≡ **if** ip(dameUno(cc)) = s **then**
  dameUno(cc)
else
  auxDameCompu(s, sinUno(cc))
fi

dameSecuDeHostnames($secu$) ≡ **if** vacia?($secu$) **then**
  <>
else
  ip(prim($secu$)) • dameSecuDeHostnames(fin($secu$))
fi

IDpaqueteEnTransito?(d, p) ≡ auxIDpaqueteEnTransito(d, computadoras(red(d)), p)

auxIDpaqueteEnTransito(d, cc, p) ≡ **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) ≡ **if** vacio?(cp) **then**
  ∅
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`)

▷ El dicc_{UNIV} (basado en un TRIE) nos permite acceder a un significado en O(L) con L el largo del hostname mas largo (utilizando como clave a los hostnames).
▷ Al guardar un iterador a la computadora con mas envíos podemos devolverla por aliasing en O(1), cumpliendo así la complejidad pedida.
▷ Al utilizar una cola_{LOG} (basada en un HEAP) podemos acceder al paquete con prioridad mas alta (el que se tiene que enviar) en O(1) y desencolarlo en O(log(n)) con n = cantidad de paquetes en la cola. Esto nos sirve para poder cumplir avanzarSegundo y nos mantiene dentro de lo pedido en crearPaquete.
▷ Al utilizar un dicc_{LOG} (basado en AA-TREE) podemos acceder a un paquete por medio de su ID en O(log(n)) con n = cantidad de paquetes en la computadora (pudiendo borrarlo o modificarlo dentro de la misma complejidad). Esto nos sirve para cumplir caminoRecorrido, ya que podemos buscar un paquete en O(log(n)) dentro de cada computadora, además nos mantiene dentro de la complejidad pedida en crearPaquete y avanzarSegundo.
▷ Al tener cantEnvios nos permite obtener en O(1) la cantidad de envíos de cada computadora, lo que nos sirve para (en avanzarSegundo) poder calcular la computadora con más envíos dentro de la complejidad y almacenarlo en conMasEnvios
▷ El índice nos sirve para (como comentamos antes) utilizarlo como posición en el arreglo caminos y poder averiguar su camino mínimo en la complejidad pedida.
▷ Guardamos los paquetes en el conjunto paquetes para poder tenerlos en O(1) y cumplir con la complejidad de enEspera.

\\ 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.

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 $\wedge$ obtener(e.computadoras, c).indice = ordenLexicografico(c, claves(e.computadoras)) $\wedge$
\\ *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

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 $\wedge$
($\forall$ 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 $\wedge$
($\forall$ 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                                                        ▷ O(1)
end function                                                           ▷ O(1)
```

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                                                           ▷ O( n * log(k) )
```

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                                                           ▷ O(L)
```

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                                                             ▷ O(L)
```

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)
\\ creo una lista vacía donde voy a guardar los hostnames y ordenarlos
    listaComp ← Vacia()                                                  ▷ O(1)
    itHostname ← CrearIt(Computadoras(r))                                ▷ O(1)
    masEnvios ← Siguiente(itHostname)                                    ▷ O(1)
    while HaySiguiente(itHostname) do                                    ▷ O(n) + O(1)
\\ agrego el hostname a la lista de computadoras
        AgregarAtras(listaComp, Siguiente(itHostname))                   ▷ O(L)
\\ Inicia el índice como cero, mas adelante les pondremos valor
        X ← <0, Vacio(), Vacio(), Vacio(), 0>                            ▷ O(1) + O(1) + O(1) + O(1) + O(1)
        itX ← DefinirRapido(diccCompus, Siguiente(itHostname), X) ▷ O(copy(hostname)) + O(copy(X))
        Definir(diccHostname, Siguiente(itHostname), itX)                ▷ O(L) + O(copy(X))
        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
    while HaySiguiente(itPC) do                                          ▷ O(#Computadoras(r))
        arrayDestinos ← CrearArreglo(n)                                  ▷ 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

\\ inicio el Indice en 0
    indice ← 0                                                                      ▷ O(1)
    while indice <#Claves(Computadoras(r)) do    ▷ Guarda: O(n)    ▷ se ejecuta n veces    ▷ O(n)
\\ busco el mínimo de la lista de hostnames (por órden alfabético
        itHostnames ← CrearIt(listaComp)                                            ▷ O(1)
        min ← Copiar(Siguiente(itHostnames))                                        ▷ O(L)
        Avanzar(itHostnames)                                                        ▷ O(1)
        while HaySiguiente(itHostnames) do
            if min <Siguiente(itHostnames) then                                     ▷ O(L)
                min ← Copiar(Siguiente(itHostnames))                                ▷ O(L)
            end if
            Avanzar(itHostnames)                                                    ▷ O(1)
        end while
        Significado(diccCompus, min).indice = indice                                ▷ O(n)
\\ creo un iterador de la lista para eliminar el minimo que ya use
        itHostnames ← CrearIt(listaComp)                                            ▷ O(1)
        noElimine ← TRUE                                                            ▷ O(1)
        while HaySiguiente(itHostnames) && noElimine do    ▷ se ejecuta a lo sumo n veces    ▷ O(n)
            if Siguiente(itHostnames) == min then                                   ▷ O(L)
                EliminarSiguiente(itHostnames)                                      ▷ O(1)
                noElimine ← FALSE                                                   ▷ O(1)
            end if
            Avanzar(itHostnames)                                                    ▷ O(1)
        end while
        indice ← indice + 1                                                         ▷ 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)
    itPaq ← AgregarRapido(SiguienteSignificado(itPC).paquetes, p)    ▷ O(copiar(nat) + O(L))    ▷ O(L)
    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                                                                      ▷ O( L + log(k) )
```

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 prioridad 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                                                        ▷ O( n * ( L + log(k) ) )
```

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                                                                         ▷ O(n * k)
```

Algorithm 23 Implementación de LaQueMasEnvió

```
function iLaQueMasEnvió(in d: estr_dcnet)→ res: hostname
    res ← SiguienteClave(d.conMasEnvios)                                                 ▷ O(1)
end function                                                                             ▷ O(1)
```

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(n*n * ( L + n*n + m ) + n*m*m)
    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(L)
            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(n * log(k))
                avanzar(itpaq)                                                           ▷ O(1)
            end while
            avanzar (itCompu)                                                            ▷ O(1)
        end while
    end if
end function                               ▷ O(n*n * ( L + n*n + m + log(k) ) + n*(m*m + L))
```

3. Módulo AB

Interfaz

se explica con: AB(σ).

géneros: `ab`(σ).

Operaciones

NIL?(**in** a: `ab`(σ)) $\rightarrow$ res : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ nil?(a)}
Complejidad: O(1)
Descripción: Indica si el arbol está vacío.

RAIZ(**in** a: `ab`(σ)) $\rightarrow$ res : σ
Pre $\equiv$ {$\neg$nil?(a)}
Post $\equiv$ {res $=_{\text{obs}}$ raiz(a)}
Complejidad: O(1)
Descripción: Devulve el valor de la raiz del arbol.
Aliasing: El valor se devuelve por referencia, res es modificable.

IZQ(**in** a: `ab`(σ) $\rightarrow$ res : `ab`(σ)
Pre $\equiv$ {$\neg$nil?(a)}
Post $\equiv$ {res $=_{\text{obs}}$ vacio()}
Complejidad: O(1)
Descripción: Devulve el subÃ¡rbol izquierdo.
Aliasing: El subÃ¡rbol se devuelve por referencia, res es modificable.

SETEARIZQ(**in/out** a: `ab`(σ, **in** i: `ab`(σ)
Pre $\equiv$ {$\neg$nil?(a)}
Post $\equiv$ {res $=_{\text{obs}}$ bin(i, raiz(a), der(a))}
Complejidad: O(1)
Descripción: Asigna i como nuevo subÃ¡rbol izquierdo.

DER(**in** a: `ab`(σ) $\rightarrow$ res : `ab`(σ)
Pre $\equiv$ {$\neg$nil?(a)}
Post $\equiv$ {res $=_{\text{obs}}$ vacio()}
Complejidad: O(1)
Descripción: Devulve el subÃ¡rbol derecho.
Aliasing: El subÃ¡rbol se devuelve por referencia, res es modificable.

SETEARDER(**in/out** a: `ab`(σ, **in** d: `ab`(σ)
Pre $\equiv$ {$\neg$nil?(a)}
Post $\equiv$ {res $=_{\text{obs}}$ bin(izq(a), raiz(a), d)}
Complejidad: O(1)
Descripción: Asigna d como nuevo subÃ¡rbol derecho.

NIL() $\rightarrow$ res : `ab`(σ)
Pre $\equiv$ {}
Post $\equiv$ {res $=_{\text{obs}}$ nil()}
Complejidad: O(1)
Descripción: Devuelve un árbol binario vacio.

BIN(**in** i: `ab`(σ), **in** d: `ab`(σ), **in** a: σ) $\rightarrow$ res : `ab`(σ)
Pre $\equiv$ {}

Post $\equiv$ {res $=_{\text{obs}}$ bin(i,d,a)}
Complejidad: O(copiar(σ))
Descripción: Crea un árbol binario con raiz a, hijo derecho d e hijo izquierdo i.

Representación

`ab(`σ`)` **se representa con** `puntero(nodo(`σ`))`

donde `nodo(`α`)` es `tupla(`*izq*: `ab(`σ`)`
der: `ab(`σ`)`
valor: σ
)

\\\\ Rep en castellano:
\\\\ 1: Si un elemento es el hijo (izquierdo o derecho) de otro, entonces no es el hijo de ningún otro
\\\\ 2: Si un elemento es hijo izquierdo de cierto elemento, no puede ser también el derecho

Rep: AB$(\alpha, \sigma) \rightarrow$ bool Rep$(estr) \equiv$ true $\iff$

\\\\ 1 $((\forall n_1, n_2, n_3 : nodo(\alpha,\sigma))((n_1 \in arbol(estr.primero) \land n_2 \in arbol(estr.primero) \land n_3 \in arbol(estr.primero) \land (n_1 = n_2 \rightarrow izq \lor n_1 = n_2 \rightarrow der) \land n_2 \neq n_3) \Rightarrow_L (n_1 \neq n_3 \rightarrow izq \land n_1 \neq n_3 \rightarrow der)) \land$

\\\\ 2 $((\forall n_1, n_2 : nodo(\alpha,\sigma))((n_1 \in arbol(estr.primero) \land n_2 \in arbol(estr.primero) \land n_1 = n_2 \rightarrow izq) \Rightarrow_L n_1 \neq n_2 \rightarrow der)$

Especificación de las operaciones auxiliares utilizadas para Rep y Abs

arbol : nodo$(\alpha \times \sigma)$ $\longrightarrow$ conj(nodo(α, σ))

caminoHastaRaiz : nodo$(\alpha \times \sigma)$ $\longrightarrow$ nat

arbol(n) $\equiv$ **if** $n.izq \neq null \land n.der \neq null$ **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

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

Algoritmos

Algorithm 25 Implementación de nil?

```
function iNil?(in a: ab(σ))→ res: bool
    res ← a == NULO)                                  ▷ O(1)
end function
```

Algorithm 26 Implementación de raiz

```
function iRaiz(in a: ab(σ))→ res: σ
    res ← a→raiz                                      ▷ O(1)
end function
```

Algorithm 27 Implementación de izq

```
function iIzq(in a: ab(σ))→ res: ab(σ)
    res ← a→izq                                        ▷ O(1)
end function
```

Algorithm 28 Implementación de setearIzq

```
function iSetearIzq(inout a: ab(σ), in i:ab(σ))
    a→izq ← i                                          ▷ O(1)
end function
```

Algorithm 29 Implementación de der

```
function iDer(in a: ab(σ))→ res: ab(σ)
    res ← a→der                                        ▷ O(1)
end function
```

Algorithm 30 Implementación de setearDer

```
function iSetearDer(inout a: ab(σ), in d:ab(σ))
    a→der ← d                                          ▷ O(1)
end function
```

Algorithm 31 Implementación de nil

```
function iNil→ res: ab(σ)
    res ← NULL                                         ▷ O(1)
end function
```

Algorithm 32 Implementación de bin

```
function iBin(in i: ab(σ), in a: σ, in d: ab(σ))→ res: ab(σ)
    res→izq ← i                                        ▷ O(1)
    res→der ← d                                        ▷ O(1)
    res→raiz ← a                                       ▷ O(copiar(σ))
end function
```

4. 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)	{¬ vacía?(c)}
desencolar	: colaPrior(α) c	$\longrightarrow$	colaPrior(α)	{¬ 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) $\vee_L$ $\Pi_1(proximo(c)) < p$ **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) $\vee_L$ $\Pi_1(proximo(c)) < p$ **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`
arb: *ab*(`tupla`(*prio*: `nat`, *valor*: α, *padre*: `ab`(α)))
)

\\ De aquí en adelante, por cuestiones de brevedad y legibilidad, se referirá a estr_heap.arb.primero simplemente como estr.primero; al padre de cada nodo (nodo.valor.padre) simplemente como nodo.padre; a el valor propiamente dicho (nodo.valor.valor) como nodo.valor; a cada nodo(tupla(nat, tupla(puntero(nodoα), α))) como nodo(α).

\\ Rep en castellano:
\\ El invariante del árbol binario se sigue cumpliendo para arb
\\ 1: El tamaño del árbol (size) debe ser igual a la cantidad de nodos en el árbol
\\ 2: El primer elemento no tiene padre
\\ 3: Todos los nodos, con la excepción del primero, deben tener padre, y deben ser el hijo izquierdo o (exclusivo) derecho de su padre (el invariante del árbol binario garantiza que ningún nodo pueda ser tanto el hijo izquierdo como el derecho de su padre)
\\ 4: La prioridad del padre es menor o igual a la de los hijos
\\ 5: La altura del árbol es igual a uno más la parte entera del logaritmo de su tamaño (es decir que la altura del árbol es O(log(n))

Rep: estr_heap(α) $\rightarrow$ bool
Rep($estr$) $\equiv$ true $\iff$
\\ 1 $size = \#arbol(estr.primero) \wedge_L$
\\ 2 $((estr.primero).padre = null \wedge$
\\ 3 $(\forall n : nodo(\alpha))(n \in arbol(estr.primero) \wedge \ n \neq estr.primero \Rightarrow (n.padre \neq null \wedge_L (((n.padre).izq = n \vee (n.padre).der = n)))) \wedge$
\\ 4 $(\forall n : nodo(\alpha))(n \in arbol(estr.primero) \Rightarrow ((n.izq \neq null \Rightarrow n.prio \leq (n.izq).prio) \wedge (n.der \neq null \Rightarrow n.prio \leq (n.der).prio))) \wedge$
\\ 5 $(\forall n : nodo(\alpha))(n \in arbol(estr.primero) \Rightarrow caminoHastaRaiz(n, arbol(estr.primero)) \leq \lfloor log_2(size) \rfloor + 1))$

Abs: estr_heap(α) e $\rightarrow$ colaPrio(α) { Rep(e) }
Abs(e) $\equiv$ c: colaPrio(α) | $(Vacia?(c) \iff e.primero == NULO) \wedge_L$
$(\neg Vacia?(c) \Rightarrow Proximo(c) = < *(estr.primero)).prioridad, *(estr.primero)).valor > \ \wedge$
$(\neg Vacia?(e) \Rightarrow Desencolar(e) = Desencolar(c))$

\\ Las operaciones auxiliares utilizadas en la especificación del Rep y el Abs están detalladas en la Estructura del Arbol Binario

Algoritmos

Algorithm 33 Implementación de Vacia

function IVACIA→ res: colaLog(α)
  res ← <0, nil()> ▷ O(1)
end function

Algorithm 34 Implementación de Vacia?

function IVACIA?(**in** $estr$: `estr_heap(`α`)`)→ res: bool
  res ← nil?(estr.arb) ▷ O(1)
end function

Algorithm 35 Implementación de Próximo

function PRÓXIMO(**in** $estr$: `estr_heap(`α`)`)→ res: tupla(nat, α)
  res ← raiz(estr) ▷ O(copiar(α))
end function

Algorithm 36 Implementación de Encolar

function iEncolar(**in/out** *estr*: `estr_heap(`α`)`, **in** *prio*: `nat`, **in** *valor*: α)$\rightarrow$ *res*: bool
  res $\leftarrow$ true ▷ O(1)
  if nil?(estr) **then** ▷ O(1)
    estr.arb $\leftarrow$ puntero(bin(null, null,<prio, valor, null>) ▷ O(copiar(α))
  else
    size++ ▷ 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.arb ▷ 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** izq(z) **else** der(z) ▷ O(1)
      y $\leftarrow$ fin(y) ▷ O(1)
    end while
    w $\leftarrow$ bin(null, null, <prio,valor, z> ▷ O(copiar(α))
    if prim(y) == 0 **then** ▷ O(1)
      setearIzq(z, w) ▷ O(1)
    else
      setearDer(z, w) ▷ O(1)
    end if
    while w $\neq$ estr.abs $\wedge_L$ π_1(raiz(w)) <π_1(raiz(π_3(raiz(w)))) **do** ▷ La cantidad de veces que se ejecuta el ciclo es a lo sumo la altura del heap, que es O(log(n))
      aux $\leftarrow$ π_1(raiz(w)) ▷ O(copiar(α))
      π_2(raiz(w)) $\leftarrow$ π_2(raiz(π_3(raiz(w)))) ▷ O(copiar(α))
      π_2(raiz(π_3(raiz(w)))) $\leftarrow$ aux ▷ O(copiar(α))
      aux2 $\leftarrow$ π_1(raiz(w)) ▷ O(1)
      π_1(raiz(w)) $\leftarrow$ π_1(raiz(π_3(raiz(w)))) ▷ O(1)
      π_1(raiz(π_3(raiz(w)))) $\leftarrow$ aux2 ▷ O(1)
      w $\leftarrow$ π_3(raiz(w)) ▷ O(1)
    end while
  end if
end function

Algorithm 37 Implementación de Desencolar

function IDESENCOLAR(**in/out** *estr* : `estr_heap`(α))→ *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 $(z.izq \neq null \ \vee z.der \neq null) \wedge_L z.prio > minPrio(z.izq, z.der)$ **do** ▷ La cantidad de veces que se ejecuta el ciclo es a lo sumo la altura del heap, que es O(log(n))

\\\\ minPrio devuelve la mínima prioridad entre los nodos 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

5. 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.*

6. 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** *arb*: *ab*(*clave*: κ, *nivel*: `nat`, *significado*: σ)

\\\\ De aquí en adelante por cuestiones de brevedad y legibilidad se referirá al primer elemento del árbol binario (arb.primero) simplemente como primero; al nivel de cada nodo (nodo.prio.nivel) como nodo.nivel; a la clave de cada nodo (nodo.prio.clave) como nodo.clave; al significado de cada nodo (nodo.valor) como nodo.significado; a cada nodo(tupla(κ, nat), σ) como nodo(κ, σ.

\\\\ La estructura utilizada para representar al diccionario Logaritmico es un AA tree. Es un tipo de ABB auto-balanceado que provee busqueda, insercion y borrado en tiempo logaritmico. Los AA trees son similares a los Red-Black Trees, pero solo pueden tener hijos derechos "rojos" (en vez de utilizar un valor booleano de color, usan un valor entero de nivel; los hijos "rojos" son los que tienen mismo nivel que sus padres), lo que reduce considerablemente la cantidad de operaciones necesarias para mantener el arbol.
\\\\ http://user.it.uu.se/∼arnea/abs/simp.html

\\\\ Rep en castellano:
\\\\ Se sigue cumpliendo el invariante del Árbol Binario
\\\\ 1: Las claves de todos los elementos del sub-árbol izquierdo de un nodo son menores que la suya. Las claves de todos los elementos del sub-árbol derecho de un nodo son mayores que la suya.
\\\\ 2: El nivel de toda hoja es 1.
\\\\ 3: El nivel de cada hijo izquierdo es exactamente 1 menos que el de su padre.
\\\\ 4: El nivel de cada hijo derecho es igual o 1 menos que el de su padre.
\\\\ 5: El nivel de cada nieto derecho (hijo derecho del hijo derecho de un nodo) es estrictamente menor que el de su abuelo.
\\\\ 6: Cada nodo con nivel (estrictamente) mayor a 1 tiene dos hijos.

Rep: arb $\rightarrow$ bool
Rep(e) $\equiv$ true $\Longleftrightarrow$
\\\\ 1 $((\forall n_1, n_2 : \ nodo(\kappa,\sigma))(\ n_1 \in \ arbol(e) \wedge \ n_2 \in \ arbol(e) \Rightarrow_L \ (n_1 \in arbol(n_2.izquierdo) \Rightarrow n_1.clave < n_2.clave) \wedge \ (n_1 \in arbol(n_2.derecho) \Rightarrow n_1.clave > n_2.clave) \wedge$
\\\\ 2 $((\forall n : \ nodo(\kappa,\sigma))(\ n \in \ arbol(e) \wedge n.izquierdo = NULO \wedge n.derecho = NULO \Rightarrow \ n.nivel = 1) \wedge$
\\\\ 3 $((\forall n : \ nodo(\kappa,\sigma))(\ n \in \ arbol(e) \wedge n.izquierdo \neq NULO \Rightarrow \ (n.izquierdo).nivel = n.nivel - 1) \wedge$
\\\\ 4 $((\forall n : \ nodo(\kappa,\sigma))(\ n \in \ arbol(e) \wedge n.derecho \neq NULO \Rightarrow ((n.derecho).nivel = n.nivel - 1 \vee (n.derecho).nivel = n.nivel)) \wedge$
\\\\ 5 $((\forall n : \ nodo(\kappa,\sigma))(\ n \in \ arbol(e) \wedge n.derecho \neq NULO \wedge_L (n.derecho).derecho \neq NULO \Rightarrow_L ((n.derecho).derecho).nivel < n.nivel) \wedge$
\\\\ 6 $((\forall n : \ nodo(\kappa,\sigma))(\ n \in \ arbol(e) \wedge n.nivel > 1 \Rightarrow \ (n.izquierdo \neq NULO \wedge n.derecho \neq NULO)$

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

Especificación de las operaciones auxiliares utilizadas para Rep y Abs

arbol: $puntero(nodo(\kappa\,\sigma)) \rightarrow conj(puntero(nodo(\kappa,\sigma)))$

$$
\begin{array}{l}
\text{arbol(n)} \;\equiv\; \textbf{if } n.izq \neq null \wedge n.der \neq null \;\; \textbf{then} \\
\qquad\qquad \text{Ag}(\&n, arbol(n.izq) \cup arbol(n.der)) \\
\qquad\quad \textbf{else} \\
\qquad\qquad \textbf{if } n.izq \neq null \;\; \textbf{then} \\
\qquad\qquad\qquad \text{Ag}(\&n, arbol(n.izq)) \\
\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad \textbf{if } n.der \neq null \textbf{ then } \text{Ag}(\&n, arbol(n.der)) \textbf{ else } \text{Ag}(\&n, \emptyset) \textbf{ fi} \\
\qquad\qquad \textbf{fi} \\
\qquad\quad \textbf{fi}
\end{array}
$$

Algoritmos

Algorithm 38 Implementación de Definido?

```
function iDefinido?(in d: estr , in c: κ)→ res: bool
\\ Búsqueda estándar en un ABB
    nodoActual ← d                                                      ▷ O(1)
    res ← FALSE                                                         ▷ O(1)
    while ¬nil?(nodoActual) && ¬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 π1(raiz(nodoActual)) == c then                               ▷ O(comparar(κ))
            res ← TRUE                                                  ▷ O(1)
        else
            if c <π1(raiz(nodoActual)) then                             ▷ O(comparar(κ))
                nodoActual ← izq(nodoActual)                            ▷ O(1)
            else
                nodoActual ← der(nodoActual)                            ▷ O(1)
            end if
        end if
    end while
end function
```

Algorithm 39 Implementación de Significado

```
function iSignificado(in d: estr , in c: κ)→ res: σ
\\ Búsqueda estándar en un ABB
    nodoActual ← d                                                      ▷ O(1)
    while ¬nil?(nodoActual) && ¬res do          ▷ El ciclo se ejecuta en el peor caso O(log(n)) veces.
        if π1(raiz(nodoActual)) == c then                               ▷ O(comparar(κ))
            res ← π3(raiz(nodoActual))     ▷ O(copiar(σ)). Esta operacion solo se ejecuta una vez (implica
¬guarda del ciclo que la contiene).
        else
            if c <π1(raiz(nodoActual)) then                             ▷ O(comparar(κ))
                nodoActual ← izq(nodoActual)                            ▷ O(1)
            else
                nodoActual ← der(nodoActual)                            ▷ O(1)
            end if
        end if
    end while
end function
```

Algorithm 40 Implementación de Vacio

function IVACIO→ res: estr
  res ← nil() ▷ O(1)
end function

Algorithm 41 Implementación de Definir

function IDEFINIR(inout d: estr , in c: κ, in s: σ)
\\ Si ya se llego a una hoja, se inserta el nuevo elemento
  if nil?(d) **then** ▷ O(1)
    $res \leftarrow < c, NULO, NULO, s, 1 >$ ▷ O(max(copiar(κ), copiar(σ))
\\ Se busca la posicion correspondiente al nuevo nodo (antes de rebalancear el arbol).
  else if c <π_1(raiz(d)) **then** ▷ O(comparar(κ)
    setearIzq(d, iDefinir(izquierdo(d),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 >π_1(raiz(d)) **then** ▷ O(comparar(κ))
    setearDer(d, 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
\\ Se tuerce y divide el arbol en cada nivel, rebalanceandolo.
  d ← Torsion(d) ▷ O(1)
  d ← Division(d) ▷ O(1)
end function

Algorithm 42 Implementación de Torsion

function ITORSION(in d: estr)→ res: estr
\\ Si el nodo tiene un hijo izquierdo del mismo nivel se debe realizar una rotacion para restaurar el invariante.
  if nil?(d) ‖ nil?(izq(d)) **then** ▷ O(1)
    res ← d ▷ O(1)
  else
\\ El hijo izquierdo de mismo nivel pasa a ser el padre del nodo derecho. El hijo derecho del nodo izquierdo pasa a ser el hijo izquierdo del nodo derecho.
    if π_2(raiz(izq(d))) == π_2(d) **then** ▷ O(1)
      nodoAux ← izq(d) ▷ O(1)
      setearIzq(d, der(nodoAux)) ▷ O(1)
      setearDer(nodoAux, d) ▷ O(1)
      res ← nodoAux ▷ O(1)
    else
      res ← d ▷ O(1)
    end if
  end if
end function

Algorithm 43 Implementación de Division

```
function iDivision(in d: estr)→ res: estr
\\ Si hay dos hijos derechos del mismo nivel que el padre se debe realizar una rotacion para restaurar el invariante.
    if nil?(d) || nil?(der(d)) || nil?(der(der(d))) then        ▷ O(1)
        res ← d                                                  ▷ O(1)
    else
\\ El primero hijo derecho pasa a ser el padre, con un nivel mas. Su hijo izquierdo pasa a ser el hijo derecho de su padre original.
        if (d.derecho.derecho).nivel == d.nivel then             ▷ O(1)
            nodoAux ← der(d)                                     ▷ O(1)
            setearDer(d, izq(nodoAux))                           ▷ O(1)
            setearIzq(nodoAux, d)                                ▷ O(1)
            π2(raiz(nodoAux)) ← π2(raiz(nodoAux))++              ▷ O(1)
            res ← nodoAux                                        ▷ O(1)
        else
            res ← d                                              ▷ O(1)
        end if
    end if
end function
```

Algorithm 44 Implementación de Borrar

function IBORRAR(inout d: estr, in c: κ)
  if nil?(d) **then** ▷ O(1)
    endFunction
\\\\ *Se busca recursivamente la posicion del elemento a borrar mediante una busqueda estandar en ABB.*
  else if c $>\pi_1$(raiz(d)) **then** ▷ O(comparar(κ))
    setearDer(d, iBorrar(der(d), 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 $<\pi_1$(raiz(d)) **then** ▷ O(comparar(κ))
    setearIzq(d, iBorrar(izq(d), 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)).
\\\\ *Si el elemento a borrar es una hoja, simplemente se lo borra.*
  else if nil?(izq(d)) $\wedge$ nil?(der(d)) **then** ▷ O(1)
    borrar(*d*) ▷ O(max(borrar(κ), borrar(σ)))
    d ← *NULO* ▷ O(1)
\\\\ *Si el elemento a borrar no es una hoja, se reduce al caso hoja.*
  else if nil?(izq(d)) **then** ▷ O(1)
\\\\ *Se busca el sucesor del elemento (bajando una vez por la rama izquierda y luego por la derecha hasta encontrar una hoja).*
    aux ← der(d) ▷ O(1)
    while ¬nil?(izq(aux)) **do** ▷ En el peor caso el ciclo se ejecuta O(log(n)) veces.
      aux ← izq(aux) ▷ O(1)
    end while
\\\\ *Se hace un swap y se elimina el elemento.*
    setearDer(d, iBorrar(π_1(raiz(aux)), der(d)))
    π_1(raiz(d)) ← π_1(raiz(aux)) ▷ O(copiar(κ))
    π_3(raiz(d)) ← π_3(raiz(aux)) ▷ O(copiar(σ))
  else
\\\\ *Se busca el predecesor del elemento (bajando una vez por la rama derecha y luego por la izquierda hasta encontrar una hoja).*
    aux ← izq(d) ▷ O(1)
    while ¬nil?(der(aux)) **do** ▷ En el peor caso el ciclo se ejecuta O(log(n)) veces.
      aux ← der(aux) ▷ O(1)
    end while
\\\\ *Se hace un swap y se elimina el elemento.*
    setearIzq(d, iBorrar(π_1(raiz(aux)), izq(d)))
    π_1(raiz(d)) ← π_1(raiz(aux)) ▷ O(copiar(κ))
    π_3(raiz(d)) ← π_3(raiz(aux)) ▷ O(copiar(σ))
  end if
\\\\ *Se nivela, divide y tuerce para restaurar el invariante.*
  d ← Nivelar(T) ▷ O(1)
  d ← Torsion(T) ▷ O(1)
  if ¬nil?(der(d)) **then** ▷ O(1)
    setearDer(der(d), Torsion(der(der(d))) ▷ O(1)
  end if
  d ← Division(T) ▷ O(1)
  setearDer(d, Division(der(d)) ▷ O(1)
  res ← d ▷ O(1)
end function
procedure NIVELAR(inout d: estr)
  nivel_correcto ← min(π_2(raiz(izq(d))), π_2(raiz(der(d))))+1 ▷ O(1)
  if nivel_correcto $<\pi_2$(raiz(d)) **then** ▷ O(1)
    π_2(raiz(d)) ← nivel_correcto ▷ O(1)
    if nivel_correcto $<\pi_2$(raiz(der(d))) **then** ▷ O(1)
      π_2(raiz(der(d))) ← nivel_correcto ▷ O(1)
    end if
  end if
end procedure