

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	3
2. Módulo DCNet	16
3. Módulo AB	28
4. Estructura Arbol Binario (α, σ)	31
5. Módulo Cola de Prioridad Logaritmica (α)	32
6. Módulo Diccionario Universal (σ)	38
7. Módulo Diccionario Logaritmico (κ, σ)	39

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** $c1$: hostname, **in** $i1$: interfaz **in** $c2$: hostname, **in** $i2$: interfaz)

Pre $\equiv \{r = r_0 \wedge c1, c2 \in dameHostnames(computadoras(r)) \wedge c1 \neq c2 \wedge$

$\neg conectadas?(r, dameCompu(c1), dameCompu(c2)) \wedge \neg usaInterfaz?(r, dameCompu(c1), i1) \wedge \neg usaInterfaz?(r, dameCompu(c2), i2) \wedge i1 \in dameCompu(c1).interfaces \wedge i2 \in dameCompu(c2).interfaces\}$

Post $\equiv \{r =_{obs} (conectar(r_0, dameCompu(c1), i1, dameCompu(c2), i2))\}$

Complejidad: O(n⁶ + n⁵ * 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.

$\text{USAIINTERFAZ?}(\text{in } r : \text{red}, \text{in } c : \text{hostname}, \text{in } i : \text{interfaz}) \rightarrow res : \text{bool}$

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

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

Complejidad: $O(n)$

Descripción: indica si la interfaz está siendo utilizada.

$\text{CAMINOSMINIMOS}(\text{in } r : \text{red}, \text{in } c_1 : \text{hostname}, \text{in } c_2 : \text{hostname}) \rightarrow res : \text{conj}(\text{lista}(\text{hostname}))$

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

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

Complejidad: $O(n*L)$

Descripción: devuelve los conjuntos de caminos minimos entre las computadoras ingresadas.

Aliasing: res se devuelve por copia.

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

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

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

Complejidad: $O(n*n)$

Descripción: indica si las computadoras son alcanzables mediante algún camino.

$\bullet == \bullet(\text{in } r_1 : \text{red}, \text{in } r_2 : \text{red}) \rightarrow res : \text{bool}$

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

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.

$\text{COPIAR}(\text{in } r : \text{red}) \rightarrow res : \text{red}$

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

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) \rightarrow conj(hostname)

dameCompu : red $r \times$ hostname $s \rightarrow$ compu $\{s \in \text{hostnames}(r)\}$

auxDameCompu : red $r \times$ hostname $s \times$ conj(compu) $cc \rightarrow$ compu $\{s \in \text{hostnames}(r) \wedge cc \subset \text{computadoras}(r)\}$

dameCaminosDeHostnames : conj(secu(compu)) \rightarrow conj(secu(hostname))

dameSecuDeHostnames : secu(compu) \rightarrow secu(hostname)

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

```

dameHostnames(cc) ≡ if vacio?(cc) then
    ∅
else
    Ag( ip(dameUno(cc)), dameHostnames(sinUno(cc)) )
fi
dameCompu(r, s) ≡ auxDameCompu(r, s, computadoras(r))
auxDameCompu(r, s, cc) ≡ if ip(dameUno(cc)) = s then
    dameUno(cc)
else
    auxDameCompu(r, s, sinUno(cc))
fi
dameCaminosDeHostnames(cs) ≡ if vacio?(cs) then
    ∅
else
    Ag(
        dameSecuDeHostnames(dameUno(cs)),
        dameCaminosDeHostnames(sinUno(cs)) )
fi
dameSecuDeHostnames(secu) ≡ 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 caminos 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, $c \in \text{claves}(e)$) (

// 1 $\text{claves}(\text{obtener}(e, c).\text{conexiones}) \subseteq \text{obtener}(e, c).\text{interfaces} \wedge$

($\forall i$: interfaz, $i \in \text{claves}(\text{obtener}(e, c).\text{conexiones})$) (

// 2 $\text{obtener}(\text{obtener}(e, c).\text{conexiones}, i) \in \text{claves}(e) \wedge$

// 3 $\text{obtener}(\text{obtener}(e, c).\text{conexiones}, i) \neq c \wedge$

// 4 ($\neg \exists i'$: interfaz, $i' \in \text{claves}(\text{obtener}(e, c).\text{conexiones})$, $i \neq i'$) $\text{obtener}(\text{obtener}(e, c).\text{conexiones}, i) == \text{obtener}(\text{obtener}(e, c).\text{conexiones}, i')$ \wedge

// 5 ($\forall h$: hostname) ($h == \text{obtener}(\text{obtener}(e, c).\text{conexiones}, i) \Rightarrow (\exists i': \text{int}) \text{obtener}(\text{obtener}(e, h).\text{conexiones}, i') == c$) \wedge

// 6 $\text{claves}(\text{obtener}(e, c).\text{alcanzables}) \subseteq \text{claves}(e) \wedge$

($\forall a$: hostname, $a \in \text{claves}(\text{obtener}(e, c).\text{alcanzables})$) (

// 7 $a \neq c \wedge$

// 8 ($\exists s$: secu(hostname)) $\text{esCaminoVálido}(c, a, s) \wedge$

// 9 $\# \text{obtener}(\text{obtener}(e, c).\text{alcanzables}, a) > 0 \wedge_L$

($\forall \text{camino}$: secu(hostname), $\text{camino} \in \text{obtener}(\text{obtener}(e, c).\text{alcanzables}, a)$) (

// 10 $\text{esCaminoVálido}(c, a, \text{camino}) \wedge$

// 11 $\neg(\exists \text{camino}'$: secu(hostname), $\text{camino} \neq \text{camino}'$, $\text{esCaminoVálido}(c, a, \text{camino}')$)

$\text{long}(\text{camino}') < \text{long}(\text{camino})$) \wedge

// 12 $\neg(\exists \text{camino}'$: secu(hostname), $\text{camino} \neq \text{camino}'$, $\text{esCaminoVálido}(c, a, \text{camino}')$,

$\text{long}(\text{camino}) == \text{long}(\text{camino}')$) ($\text{camino}' \notin \text{obtener}(\text{obtener}(e, c).\text{alcanzables}, a)$)

))

))

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

$\text{esCaminoVálido}(\text{orig}, \text{dest}, \text{secu}) \equiv$ ($\text{prim}(\text{secu}) == \text{orig} \wedge$

($\forall i$: nat, $0 < i < \text{long}(\text{secu})$) $\text{esVecino}(\text{secu}[i], \text{secu}[i+1]) \wedge$

$\text{secu}[\text{long}(\text{secu})-1] == \text{dest} \wedge$

$\text{sinRepetidos}(\text{secu})$)

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

Abs : estr_red $e \longrightarrow$ red

{Rep(e)}

Abs(e) $\equiv r$ | $\text{computadoras}(r) = \text{dameComputadoras}(e) \wedge_L$

($\forall c1, c2$: compu, $c1, c2 \in \text{computadoras}(r)$) $\text{conectadas?}(r, c1, c2) = (\exists i$: interfaz) ($c2.\text{ip} =$

$\text{obtener}(\text{obtener}(e, c1.\text{ip}).\text{conexiones}, i)$) \wedge_L

$\text{interfazUsada}(r, c1, c2) = \text{buscarClave}(\text{obtener}(e, c1.\text{ip}).\text{conexiones}, c2.\text{ip})$

Especificación de las funciones auxiliares utilizadas en abs

$\text{dameComputadoras} : \text{dicc}(\text{hostname}; X) \longrightarrow \text{conj}(\text{computadoras})$
 $\text{auxDameComputadoras} : \text{dicc}(\text{hostname}; X) \times \text{conj}(\text{hostname}) \longrightarrow \text{conj}(\text{computadoras})$
 $\text{buscarClave} : \text{dicc}(\text{interfaz}; \text{hostname}) \times \text{hostname} \longrightarrow \text{interfaz}$
 $\text{auxBuscarClave} : \text{dicc}(\text{interfaz}; \text{hostname}) \times \text{hostname} \times \text{conj}(\text{interfaz}) \longrightarrow \text{interfaz}$

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

$\text{dameComputadoras}(e) \equiv \text{auxDameComputadoras}(e, \text{claves}(e))$
 $\text{auxDameComputadoras}(e, cc) \equiv \text{if } \emptyset?(cc) \text{ then } \emptyset$
 $\quad \text{else}$
 $\quad \text{Ag}(\text{<dameUno}(cc), \text{obtener}(e, \text{dameUno}(cc)).\text{interfaces}>, \text{auxDameComputadoras}(e, \text{sinUno}(cc)))$
 $\quad \text{fi}$

$\text{buscarClave}(d, h) \equiv \text{auxBuscarClave}(d, h, \text{claves}(d))$
 $\text{auxBuscarClave}(d, h, ci) \equiv \text{if } \text{obtener}(d, \text{dameUno}(cc)) = h \text{ then } \text{dameUno}(cc)$
 $\quad \text{else}$
 $\quad \text{auxBuscarClave}(d, h, \text{sinUno}(ci))$
 $\quad \text{fi}$

Algoritmos

Algorithm 1 Implementación de Computadoras

```

function iCOMPUTADORAS(in r: estr_red)→ res: conj(hostname)
  it ← crearIt(r)                                ▷ O(1)
  res ← Vacio()                                  ▷ 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(n4 + n3 x L)
    ActualizarCaminos(r, c2, c1)                                         ▷ O(n4 + n3 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(n6 + n5 * L)
end function                                                             ▷ O(n6 + n5 * 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 están 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
        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
        end if
      end if
    end while
  end if

```

Avanzar(itAlcanzables1)	$\triangleright O(1)$
end while	$\triangleright O(n \times (n + (n \times (n + L))))$
Avanzar(itAlcanzables2)	$\triangleright O(1)$
end while	$O(n + n \times (n \times (n + (n \times (n + L))))$
end function	$\triangleright O(n^4 + n^3 \times L)$

Algorithm 8 Implementación de función auxiliar Actualizar Vecinos

function iACTUALIZARVECINOS(inout r: estr_red, in c1: hostname, in conj(hostname) actualizados)	
<i>// Actualiza los caminos de los vecinos de C, y luego hace recursion para los vecinos de los vecinos.</i>	
itVecinos \leftarrow crearIt(Significado(r, c1).conexiones)	$\triangleright O(n)$
while (HaySiguiente(itVecinos)) do	$\triangleright O(n \times x)$
<i>// Si todavÁa no fue actualizado, lo actualizo y hago recursión sobre los vecinos.</i>	
if (SiguienteClave(itVecinos) \notin actualizados) then	$\triangleright O(n \times L)$
ActualizarCaminos(r, SiguienteClave(itVecinos), c)	$\triangleright O(n^4 + n^3 \times L)$
AgregarRapido (actualizados, SiguienteClave(itVecinos))	$\triangleright O(L)$
ActualizarVecinos(r, SiguienteClave(itVecinos), actualizados)	\triangleright recursión hasta actualizar las n
computadoras, $O(n) \times$	
end if	
Avanzar(itVecinos)	$\triangleright O(1)$
end while	
end function	$\triangleright O(n^6 + n^5 \times L)$

Algorithm 9 Implementación de Vecinos

function iVECINOS(inout r: estr_red, in c1: hostname) \rightarrow res: conj(hostname)	
it \leftarrow CrearIt(Significado(r,c1).conexiones)	$\triangleright O(1) + O(n)$
res \leftarrow Vacio()	\triangleright Conjunto $\triangleright O(1)$
while HaySiguiente(it) do	\triangleright Guarda: $O(1)$ \triangleright El ciclo se ejecuta a lo sumo n-1 veces $\triangleright O(n)$
AgregarRapido(res, SiguienteSignificado(it))	$\triangleright O(L)$
Avanzar(it)	$\triangleright O(1)$
end while	
end function	$\triangleright O(n \times L)$

Algorithm 10 Implementación de UsaInterfaz

function iUSAINTERFAZ(in r: estr_red, in c: hostname, in i: interfaz) \rightarrow res: bool	
res \leftarrow Definido?(Significado(r,c).conexiones,i)	$\triangleright O(\text{comparar}(\text{nat}) \times n)$
end function	$\triangleright O(n)$

Algorithm 11 Implementación de CaminosMinimos

function iCAMINOSMINIMOS(in r: estr_red, in c1: hostname, in c2: hostname) \rightarrow res: conj(lista(hostname))	
itCaminos \leftarrow crearIt(Significado(Significado(r,c1).alcanzables, c2))	$\triangleright O(1) + O(L \times n)$
res \leftarrow Vacio()	\triangleright Conjunto $\triangleright O(1)$
while HaySiguiente(itCaminos) do	
AgregarRapido(res, Siguiente(itCaminos))	$\triangleright O(1)$
Avanzar(itCaminos)	$\triangleright O(1)$
end while	
end function	$\triangleright O(n \times L)$

Algorithm 12 Implementación de HayCamino?

function iHAYCAMINO?(in r: estr_red, in c1: hostname, in c2: hostname) \rightarrow res: bool	
res \leftarrow Definido?(Significado(r,c1).alcanzables, c2)	$\triangleright O(n \times n)$
end function	$\triangleright O(n \times n)$

Algorithm 13 Implementación de ==

```

function IGUALDAD(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 está 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 ¬(Compu1.interfaces == Compu2.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 ¬(Compu1.conexiones == Compu2.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?(Compu2.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(Compu2.alcanzables, itAlc1) ▷ O(n)
                        end if
                    end while
                    // Me guardo los 2 conjuntos de caminos (de la compu de la red 1 y la de la red 2)
                end else
            end if
        end while
    end if
    res

```

```

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

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

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 $\equiv \{\text{IDpaqueteEnTransito?}(d, p)\}$

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

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

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

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

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

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

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

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

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

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 $\equiv \{d = d_0\}$

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

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**) $\rightarrow res$: **bool**

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

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

Complejidad: $O(n * k)$

Descripción: indica si el paquete esta en alguna de las colas dado el **ID**.

LAQUEMASENVIO(**in** d : **dcnet**) $\rightarrow res$: **hostname**

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

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

Complejidad: $O(1)$

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

Post $\equiv \{res =_{obs} (d_1 =_{obs} d_2)\}$

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 $d \times$ 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 $\forall d: \text{dcnet}, \forall s: \text{hostname}, \forall p: \text{IDpaquete}, \forall cc: \text{conj}(\text{compu}), , \forall secu: \text{secu}(\text{compu}), \forall cp: \text{conj}(\text{paquete}),$

dameHostnames(cc) \equiv **if** vacio?(cc) **then**
 \emptyset
else
 $\text{Ag}(\text{ip}(\text{dameUno}(cc)), \text{dameHostnames}(\text{sinUno}(cc)))$
fi
dameCompu(d, s) \equiv auxDameCompu($s, \text{computadoras}(\text{red}((d)))$)
auxDameCompu(s, cc) \equiv **if** ip($\text{dameUno}(cc)$) = s **then**
 $\text{dameUno}(cc)$
else
 $\text{auxDameCompu}(s, \text{sinUno}(cc))$
fi
dameSecuDeHostnames($secu$) \equiv **if** vacia?($secu$) **then**
 $\langle \rangle$
else
 $\text{ip}(\text{prim}(secu)) \bullet \text{dameSecuDeHostnames}(\text{fin}(secu))$
fi
IDpaqueteEnTransito?(d, p) \equiv auxIDpaqueteEnTransito($d, \text{computadoras}(\text{red}(d)), p$)
auxIDpaqueteEnTransito(d, cc, p) \equiv **if** vacio?(cc) **then**
false
else
if $p \in \text{dameIDpaquetes}(\text{enEspera}(\text{dameUno}(cc)))$ **then**
true
else
 $\text{auxIDpaqueteEnTransito}(d, \text{sinUno}(cc), p)$
fi
fi
dameIDpaquetes(cp) \equiv **if** vacio?(cp) **then**
 \emptyset
else
 $\text{Ag}(\text{id}(\text{dameUno}(cp)), \text{dameIDpaquetes}(\text{sinUno}(cp)))$
fi

Fin TAD

Representación

dcnet se representa con `estr_dcnet`

```

donde estr_dcnet es tupla(red: red
    computadoras: dicc(hostname, X)
    porHostname: diccUNIV (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: colaLOG(itConj(paquete))
    paqPorID: diccLOG (IDpaquete, itConj(paquete))
    cantEnvios: nat )

```

▷ El `diccUNIV` (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 `colaLOG` (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 `diccLOG` (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 índices de las computadoras van de 0 a $n-1$.

5: Los índices no se repiten.

6: `ConMasEnvios` es un iterador a la computadora con mayor `cant` de envios.

7: La matriz de `caminos` es de $n \times n$.

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 índices 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 tamaño.

11: La cola ordena los paquetes por prioridad. (usando los observadores del TAD Cola de Prioridad Alaterativa adjunto).

Para todos los paquetes de una computadora:

- 12: El origen y el destino estan entre las computadoras de la dcnnet.*
- 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 \iff

```

\\ 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))
 $\neg \quad \emptyset?$  (caminosMinimos(e.red, dameCompu(c1), dameCompu(c2)))  $\implies$ 
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)))  $\implies$ 
e.caminos[obtener(e.computadoras, c1).indice][obtener(e.computadoras, c2).indice] = Va-
cia()  $\wedge$ 
( $\forall c$ : 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( $\Pi_2$ (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( $\Pi_2$ (proximo(obtener(e.computadoras, c).cola))).prioridad  $\wedge$ 
 $\Pi_1$ (proximo(obtener(e.computadoras, c).cola)) = Siguiente( $\Pi_2$ (proximo(obtener(e.computadoras,
c).cola))).prioridad  $\wedge$ 
desencolar(obtener(e.computadoras, c).cola) = armarCola(obtener(e.computadoras, c).paquetes
- {Siguiente( $\Pi_2$ (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 $\quad \forall cc$: conj(paquete)

armarCola(cc) \equiv **if** $\emptyset?$ (cc) **then**

Vacia()

else

encolar(dameUno(cc).prioridad, dameUno(cc), armarCola(sinUno(cc)))

fi

$Abs : estr_dcnet\ e \longrightarrow dcnet \quad \{Rep(e)\}$
 $Abs(e) \equiv d \mid red(d) = e.red \wedge$
 $(\forall c: compu, c \in computadoras(red(d))) ($
 $cantidadEnviados(d, c) = obtener(e.computadoras, c.ip).cantEnvios \wedge$
 $enEspera(d, c) = obtener(e.computadoras, c.ip).paquetes \wedge$
 $(\forall p: paquete, p \in obtener(e.computadoras, c.ip).paquetes) caminoRecorrido(d, p) =$
 $e.camino[obtener(e.computadoras, origen(p).ip).indice][obtener(e.computadoras, c.ip).indice])$

Algoritmos

Algorithm 15 Implementación de Red

```

function IRED(in  $d: estr\_dcnet$ )  $\rightarrow res: Red$ 
     $res \leftarrow d.red$   $\triangleright O(1)$ 
end function  $\triangleright O(1)$ 

```

Algorithm 16 Implementación de CaminoRecorrido

```

function ICAMINORECORRIDO(in  $d: estr\_dcnet$ , in  $p: IDPaquete$ )  $\rightarrow res: lista(hostname)$ 
     $itCompu \leftarrow CrearIt(d.computadoras)$   $\triangleright O(1)$ 
     $yaEncontrado \leftarrow FALSE$   $\triangleright O(1)$ 
    while HaySiguiente( $itCompu$ ) &&  $\neg yaEncontrado$  do  $\triangleright$  Guarda:  $O(1)$   $\triangleright$  Se repite a lo sumo  $n$  veces  $\triangleright$ 
 $O(n)$ 
        if Definido?(SiguienteSignificado( $itCompu$ ).paqPorID,  $p$ ) then  $\triangleright O(\log(k))$ 
             $paquete \leftarrow Significado(SiguienteSignificado(itCompu).paqPorID, p)$   $\triangleright O(1)$ 
             $yaEncontrado \leftarrow TRUE$   $\triangleright O(1)$ 
        else
            Avanzar( $itCompu$ )  $\triangleright O(1)$ 
        end if
    end while
     $res \leftarrow caminos[Significado(d.computadoras, \pi_3(paquete)).indice][SiguienteSignificado(itCompu).indice]$ 
 $\triangleright O(1) + O(n) + O(1)$ 
end function  $\triangleright O(n * \log(k))$ 

```

Algorithm 17 Implementación de paquetes enviados

```

function ICANTIDADENVIADOS(in  $d: estr\_dcnet$ , in  $c: hostname$ )  $\rightarrow res: nat$ 
     $it \leftarrow Significado(d.porHostname, c)$   $\triangleright O(L)$ 
     $res \leftarrow SiguienteSignificado(it).cantEnvios$   $\triangleright O(1)$ 
end function  $\triangleright O(L)$ 

```

Algorithm 18 Implementación EnEspera

```

function IENESPERA(in d: estr_dcnet, in c: hostname)  $\rightarrow$  res: estr
    it  $\leftarrow$  Significado(d.porHostname, c)  $\triangleright$  O(L)
    res  $\leftarrow$  SiguienteSignificado(it).paquetes  $\triangleright$  O(1)
end function  $\triangleright$  O(L)

```

Algorithm 19 Implementación de iniciarDCNet

```

function IINICIARDCNET(in r: red)  $\rightarrow$  res: estr_dcnet
    // creo un diccionario lineal
    diccCompus  $\leftarrow$  Vacio()  $\triangleright$  O(1)
    // creo un diccionario universal(trie)
    diccHostname  $\leftarrow$  Vacio()  $\triangleright$  O(1)
    // creo una lista vacía donde voy a guardar los hostnames y ordenarlos
    listaComp  $\leftarrow$  Vacía()  $\triangleright$  O(1)
    itHostname  $\leftarrow$  CrearIt(Computadoras(r))  $\triangleright$  O(1)
    masEnvios  $\leftarrow$  Siguiente(itHostname)  $\triangleright$  O(1)
    while HaySiguiente(itHostname) do  $\triangleright$  O(n) + O(1)
        // agrego el hostname a la lista de computadoras
        AgregarAtras(listaComp, Siguiente(itHostname))  $\triangleright$  O(L)
        // Inicia el índice como cero, mas adelante les pondremos valor
        X  $\leftarrow$  <0, Vacio(), Vacio(), Vacio(), 0>  $\triangleright$  O(1) + O(1) + O(1) + O(1) + O(1)
        itX  $\leftarrow$  DefinirRapido(diccCompus, Siguiente(itHostname), X)  $\triangleright$  O(copy(hostname)) + O(copy(X))
        Definir(diccHostname, Siguiente(itHostname), itX)  $\triangleright$  O(L) + O(copy(X))
        Avanzar(itHostname)  $\triangleright$  O(1)
    end while
    itPC  $\leftarrow$  CrearIt(diccCompus)  $\triangleright$  O(1)
    itPC2  $\leftarrow$  CrearIt(diccCompus)  $\triangleright$  O(1)
    n  $\leftarrow$  #Claves(diccCompus)  $\triangleright$  O(1)
    arrayCaminos  $\leftarrow$  CrearArreglo(n)  $\triangleright$  O(n)
    // voy a crear un arreglo en cada posicion de arrayCaminos, el cual va a tener el minimo camino
    while HaySiguiente(itPC) do  $\triangleright$  O(#Computadoras(r))
        arrayDestinos  $\leftarrow$  CrearArreglo(n)  $\triangleright$  O(n)
        while HaySiguiente(itPC2) do  $\triangleright$  O(#Computadoras(r))
            ConjCaminos  $\leftarrow$  CaminosMinimos(r, SiguienteClave(itPC), SiguienteClave(itPC2))  $\triangleright$  O(n*L)
            itConj  $\leftarrow$  CrearIt(ConjCaminos)  $\triangleright$  O(1)
            // de todos los caminos minimos me quedo con uno
            if HaySiguiente(itConj) then  $\triangleright$  O(1)
                arrayDestinos[SiguienteSignificado(itPC2).indice]  $\leftarrow$  Siguiente(itConj)  $\triangleright$  O(1)
            else
                // si no hay camino, creo una lista vacia
                arrayDestinos[SiguienteSignificado(itPC2).indice]  $\leftarrow$  Vacía()  $\triangleright$  O(1)
            end if

```

Avanzar(itPC2)	▷ O(1)
end while	
arrayCaminos[SiguienteSignificado(itPC).indice] ← arrayDestinos	▷ O(1)
Avanzar(itPC)	▷ O(1)
end while	
<i>\\ inicio el Indice en 0</i>	
indice ← 0	▷ O(1)
while indice < #Claves(Computadoras(r)) do	▷ Guarda: O(n) ▷ se ejecuta n veces ▷ O(n)
<i>\\ busco el mínimo de la lista de hostnames (por orden alfabético)</i>	
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)
<i>\\ creo un iterador de la lista para eliminar el minimo que ya use</i>	
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
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_dcnnet)
  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 (¬(Vacía?(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.camino[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 LaQueMasEnvio

```

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

```

Algorithm 24 Implementación de ==

```

function IGUALDAD(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: $\text{AB}(\sigma)$.

géneros: $\text{ab}(\sigma)$.

Operaciones

$\text{NIL?}(\text{in } a : \text{ab}(\sigma)) \rightarrow res : \text{Bool}$

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

Post $\equiv \{res =_{\text{obs}} \text{nil?}(a)\}$

Descripción: Indica si el árbol está vacío.

$\text{RAIZ}(\text{in } a : \text{ab}(\sigma)) \rightarrow res : \sigma$

Pre $\equiv \{\neg \text{nil?}(a)\}$

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

Descripción: Devuelve el valor de la raíz del árbol.

Aliasing: alias?

$\text{IZQ}(\text{in } a : \text{ab}(\sigma)) \rightarrow res : \text{ab}(\sigma)$

Pre $\equiv \{\neg \text{nil?}(a)\}$

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

Descripción: Devuelve un puntero apuntando al hijo izquierdo.

$\text{DER}(\text{in } a : \text{ab}(\sigma)) \rightarrow res : \text{ab}(\sigma)$

Pre $\equiv \{\neg \text{nil?}(a)\}$

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

Descripción: Devuelve un puntero apuntando al hijo derecho.

$\text{NIL}() \rightarrow res : \text{ab}(\sigma)$

Pre $\equiv \{\}$

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

Descripción: Devuelve un árbol binario vacío.

$\text{BIN}(\text{in } i : \text{ab}(\sigma), \text{in } a : \sigma, \text{in } d : \text{ab}(\sigma)) \rightarrow res : \text{ab}(\sigma)$

Pre $\equiv \{\}$

Post $\equiv \{res =_{\text{obs}} \text{bin}(i,a,d)\}$

Descripción: Crea un árbol binario con raíz a, hijo derecho d e hijo izquierdo i.

Representación

AB se representa con $arbol_{AB}$: $tupla(raiz: \sigma, izq: arbol_{AB}, der: arbol_{AB})$

Algoritmos

Algorithm 25 Implementación de nil?

```
function INIL?(in a:  $arbol_{AB}$ )  $\rightarrow$  res: bool
  res  $\leftarrow$  (a.izq == NULO  $\wedge$  a.der == NULO)
end function
```

Algorithm 26 Implementación de raíz

```
function IRAIZ(in a:  $arbol_{AB}$ )  $\rightarrow$  res:  $\sigma$ 
  res  $\leftarrow$  a.raiz
end function
```

Algorithm 27 Implementación de izq

```
function IIZQ(in a:  $arbol_{AB}$ )  $\rightarrow$  res:  $arbol_{AB}$ 
  res  $\leftarrow$  a.izq
end function
```

Algorithm 28 Implementación de der

```
function iDER(in a:  $arbol_{AB}$ )  $\rightarrow$  res:  $arbol_{AB}$   
    res  $\leftarrow$  a.der  
end function
```

Algorithm 29 Implementación de nil

```
function iNIL  $\rightarrow$  res:  $arbol_{AB}$   
    res  $\leftarrow$  <NULO,NULO,NULO>  
end function
```

Algorithm 30 Implementación de bin

```
function iIZQ(in i:  $arbol_{AB}$ , in a:  $\sigma$ , in d:  $arbol_{AB}$ )  $\rightarrow$  res:  $arbol_{AB}$   
    res  $\leftarrow$  nil()  
    res.izq  $\leftarrow$  i  
    res.der  $\leftarrow$  d  
    res.raiz  $\leftarrow$  a  
end function
```

4. Estructura Arbol Binario (α, σ)

AB es *primero*: puntero(nodo(α, σ))

donde nodo(α) es tupla(*izq*: puntero(nodo(α, σ))
der: puntero(nodo(α, σ))
prio: α
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: $AB(\alpha, \sigma) \rightarrow \text{bool}$ Rep(*estr*) $\equiv \text{true} \iff$

// 1 $((\forall n_1, n_2, n_3 : \text{nodo}(\alpha, \sigma))((n_1 \in \text{arbol}(\text{estr.primero}) \wedge n_2 \in \text{arbol}(\text{estr.primero}) \wedge n_3 \in \text{arbol}(\text{estr.primero}) \wedge (n_1 = n_2 \rightarrow \text{izq} \vee n_1 = n_2 \rightarrow \text{der}) \wedge n_2 \neq n_3) \Rightarrow_L (n_1 \neq n_3 \rightarrow \text{izq} \wedge n_1 \neq n_3 \rightarrow \text{der})) \wedge$

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

Especificación de las operaciones auxiliares utilizadas para Rep y Abs

arbol : $\text{nodo}(\alpha \times \sigma) \rightarrow \text{conj}(\text{nodo}(\alpha, \sigma))$

caminoHastaRaiz : $\text{nodo}(\alpha \times \sigma) \rightarrow \text{nat}$

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

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

5. 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() $\rightarrow res : \text{colaLog}(\alpha)$

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

Post $\equiv \{res =_{obs} \text{vacia}\}$

Complejidad: $O(1)$

Descripción: Crea una cola vacia.

VACIA?(in *estr*: colaLog(α)) $\rightarrow res : \text{bool}$

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

Post $\equiv \{res =_{obs} \text{vacia?}(\text{estr})\}$

Complejidad: $O(1)$

Descripción: Indica si la cola esta vacia.

PRÓXIMO(in *estr*: colaLog(α)) $\rightarrow res : \text{tupla}(\text{nat}, \alpha)$

Pre $\equiv \{\neg \text{Vacía?}(\text{estr})\}$

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

Complejidad: $O(\text{copiar}(\alpha))$

Descripción: Devuelve una tupla que contiene al próximo elemento y su prioridad.

ENCOLAR(in/out *estr*: colaLog(α), in *prio*: nat, in *valor*: α) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{estr} = \text{estr}_0\}$

Post $\equiv \{res \wedge \text{estr} =_{obs} \text{encolar}(\text{estr}_0)\}$

Complejidad: $O(\log(n) + \text{copiar}(\alpha))$

Descripción: Crea un nuevo elemento con los parametros dados y lo agrega a la cola.

DESENCOLAR(in/out *estr*: colaLog(α)) $\rightarrow res : \alpha$

Pre $\equiv \{\text{estr} = \text{estr}_0 \wedge \neg \text{Vacía?}(\text{estr})\}$

Post $\equiv \{\text{estr} =_{obs} \text{desencolar}(\text{estr}_0) \wedge res =_{obs} \text{proximo}(\text{estr}_0)\}$

Complejidad: $O(\log(n) + \text{copiar}(\alpha) + \text{borrar}(\alpha))$

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)

g neros colaPrio( )

exporta	colaPrio(α), generadores, observadores
----------------	---

usa	BOOL, NAT, TUPLA
------------	------------------

observadores básicos

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

generadores

$$\text{vacía} \quad : \quad \longrightarrow \text{colaPrior}(\alpha)$$
$$\text{encolar} : \text{nat} \times \alpha \times \text{colaPrior}(\alpha) \longrightarrow \text{colaPrior}(\alpha)$$
$$\text{axioms} \quad \forall c: \text{colaPrior}(\alpha), \forall e: \alpha$$

`vacía?(vacía)` \equiv `true`

$$\text{vacía?}(\text{encolar}(p, e, c)) \quad \equiv \quad \text{false}$$
$$\text{pr\u00f3ximo}(\text{encolar}(p, e, c)) \quad \equiv \quad \text{if } \text{vac\u00eda?}(c) \vee_{\text{L}} \Pi_1(\text{pr\u00f3ximo}(c)) < p \text{ then } < p, e > \text{ else if } \\ \Pi_1(\text{pr\u00f3ximo}(c)) = p \text{ then } < p, e > \vee \text{pr\u00f3ximo}(c) \text{ else } \text{pr\u00f3ximo}(c) \text{ fi} \\ \text{fi}$$
$$\text{desencolar}(\text{encolar}(p, e, c)) \equiv \text{if } \text{vacía?}(c) \vee_{\text{L}} \Pi_1(\text{proximo}(c)) < p \text{ then } c \text{ else if } \Pi_1(\text{proximo}(c)) = p \text{ then } c \vee \text{encolar}(p, e, \text{desencolar}(c)) \text{ else } \text{encolar}(p, e, \text{desencolar}(c)) \text{ fi fi}$$

Fin TAD

Representación

`colaLog(α)` se representa con `estr_heap(α)`

donde `estr_heap(α)` es `tupla(size: nat`
`arb: argolAB(tupla(nat, valor: α))`
`)`

// De aquí en adelante, por cuestiones de brevedad y legibilidad, se referirá a `estr.arb.primer` simplemente como `estr.primer`; 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(nodo α), 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.primer) \wedge_L`
`// 2 ((estr.primer).padre = null \wedge`
`// 3 ($\forall n : \text{nodo}(\alpha)$)($n \in \text{arbol}(\text{estr.primer}) \wedge n \neq \text{estr.primer} \Rightarrow (n.padre \neq \text{null} \wedge_L (((n.padre).izq =$`
 `$n \vee (n.padre).der = n)))) \wedge$`
`// 4 ($\forall n : \text{nodo}(\alpha)$)($n \in \text{arbol}(\text{estr.primer}) \Rightarrow ((n.izq \neq \text{null} \Rightarrow n.prio \leq (n.izq).prio) \wedge (n.der \neq \text{null} \Rightarrow$`
 `$n.prio \leq (n.der).prio))) \wedge$`
`// 5 ($\forall n : \text{nodo}(\alpha)$)($n \in \text{arbol}(\text{estr.primer}) \Rightarrow \text{caminoHastaRaiz}(n, \text{arbol}(\text{estr.primer})) \leq \lfloor \log_2(\text{size}) \rfloor +$`
`1))`

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

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

Algoritmos

Algorithm 31 Implementación de Vacía

```
function iVACIA  $\rightarrow$  res: colaLog( $\alpha$ )  
  res  $\leftarrow$  nil()  
end function
```

$\triangleright O(1)$

Algorithm 32 Implementación de Vacía?

```
function iVACIA? (in estr: estr_heap( $\alpha$ ))  $\rightarrow$  res: bool  
  res  $\leftarrow$  nil?(estr)  
end function
```

$\triangleright O(1)$

Algorithm 33 Implementación de Próximo

```
function PRÓXIMO (in estr: estr_heap( $\alpha$ ))  $\rightarrow$  res: tupla(nat,  $\alpha$ )  
  res  $\leftarrow$  raiz(estr)  
end function
```

$\triangleright O(\text{copiar}(\alpha))$

Algorithm 34 Implementación de Encolar

```

function iENCOLAR(in/out estr: estr_heap( $\alpha$ ), in prio: nat, in valor:  $\alpha$ )  $\rightarrow$  res: bool
    res  $\leftarrow$  true  $\triangleright O(1)$ 
    if nil?(estr) then  $\triangleright O(1)$ 
        estr.primer  $\leftarrow$  puntero( $\langle$  null, null, null, prio, valor  $\rangle$ )  $\triangleright O(\text{copiar}(\alpha))$ 
    else
        size ++  $\triangleright O(1)$ 
        x  $\leftarrow$  sizer  $\triangleright O(1)$ 
        y  $\leftarrow$   $\langle \rangle$   $\triangleright O(1)$ 
        while x  $\neq$  0 do  $\triangleright$  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) • y  $\triangleright O(1)$ 
            x  $\leftarrow$  x / 2  $\triangleright O(1)$ 
        end while
        y  $\leftarrow$  com(y)  $\triangleright O(\log(n))$ 
        z  $\leftarrow$  estr.primer  $\triangleright O(1)$ 
        y  $\leftarrow$  fin(y)  $\triangleright O(1)$ 
        while long(y) > 1 do  $\triangleright$  El ciclo se ejecuta  $O(\log(n))$  veces
            z  $\leftarrow$  if prim(y) == 0 then izq(z) else der(z)  $\triangleright O(1)$ 
            y  $\leftarrow$  fin(y)  $\triangleright O(1)$ 
        end while
        w  $\leftarrow$   $\langle$  null, null, null, prio, valor  $\rangle$   $\triangleright O(\text{copiar}(\alpha))$ 
        w.padre  $\leftarrow$  z  $\triangleright O(1)$ 
        if prim(y) == 0 then  $\triangleright O(1)$ 
            z.izq  $\leftarrow$  w  $\triangleright O(1)$ 
        else
            z.der  $\leftarrow$  w  $\triangleright O(1)$ 
        end if
        while w  $\neq$  estr.primer  $\wedge_L$  w.prio < (w.padre).prio do  $\triangleright$  La cantidad de veces que se ejecuta el
            ciclo es a lo sumo la altura del heap, que es  $O(\log(n))$ 
            aux  $\leftarrow$  w.valor  $\triangleright O(\text{copiar}(\alpha))$ 
            w.valor  $\leftarrow$  (w.padre).valor  $\triangleright O(\text{copiar}(\alpha))$ 
            (w.padre).valor  $\leftarrow$  aux  $\triangleright O(\text{copiar}(\alpha))$ 
            aux2  $\leftarrow$  w.prio  $\triangleright O(1)$ 
            w.prio  $\leftarrow$  (w.padre).prio  $\triangleright O(1)$ 
            (w.padre).prio  $\leftarrow$  aux2  $\triangleright O(1)$ 
            w  $\leftarrow$  w.padre  $\triangleright O(1)$ 
        end while
    end if
end function

```

Algorithm 35 Implementación de Desencolar

```

function IDESENCOLAR(in/out estr: estr_heap( $\alpha$ ))  $\rightarrow$  res:  $\alpha$ 
    res  $\leftarrow$  *(estr.primer)  $\triangleright$  O(1)
    x  $\leftarrow$  size  $\triangleright$  O(1)
    y  $\leftarrow$   $\langle \rangle$   $\triangleright$  O(1)
    while x  $\neq$  0 do  $\triangleright$  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) • y  $\triangleright$  O(1)
        x  $\leftarrow$  x / 2  $\triangleright$  O(1)
    end while
    y  $\leftarrow$  com(y)  $\triangleright$  O(log(n))
    z  $\leftarrow$  estr.primer  $\triangleright$  O(1)
    y  $\leftarrow$  fin(y)  $\triangleright$  O(1)
    while long(y) > 1 do  $\triangleright$  El ciclo se ejecuta O(log(n)) veces
        z  $\leftarrow$  if prim(y) == 0 then z.izq else z.der  $\triangleright$  O(1)
        y  $\leftarrow$  fin(y)  $\triangleright$  O(1)
    end while
    w  $\leftarrow$   $\langle$  null, null, null, prio, valor  $\rangle$   $\triangleright$  O(copiar( $\alpha$ ))
    w.padre  $\leftarrow$  z  $\triangleright$  O(1)
    if prim(y) == 0 then  $\triangleright$  O(1)
        z.izq  $\leftarrow$  puntero(w)  $\triangleright$  O(1)
    else
        z.der  $\leftarrow$  puntero(w)  $\triangleright$  O(1)
    end if
    (estr.primer).valor  $\leftarrow$  z.valor  $\triangleright$  O(copiar( $\alpha$ ))
    (estr.primer).prio  $\leftarrow$  z.prio  $\triangleright$  O(1)
    borrar(z)  $\triangleright$  O(borrar( $\alpha$ ))
    z  $\leftarrow$  estr.primer  $\triangleright$  O(1)
    size  $\leftarrow$   $\text{--}$   $\triangleright$  O(1)
    while (z.izq  $\neq$  null  $\vee$  z.der  $\neq$  null)  $\wedge_L$  z.prio > minPrio(z.izq, z.der) do  $\triangleright$  La cantidad de veces
    que se ejecuta el ciclo es a lo sumo la altura del heap, que es O(log(n))

    if z.der == null  $\vee_L$  (z.izq).prio  $\geq$  (z.der).prio then  $\triangleright$  O(1)
        aux  $\leftarrow$  z.valor  $\triangleright$  O(copiar( $\alpha$ ))
        z.valor  $\leftarrow$  (z.izq).valor  $\triangleright$  O(copiar( $\alpha$ ))
        (z.izq).valor  $\leftarrow$  aux  $\triangleright$  O(copiar( $\alpha$ ))
        aux2  $\leftarrow$  z.prio  $\triangleright$  O(1)
        z.prio  $\leftarrow$  (z.izq).prio  $\triangleright$  O(1)
        (z.izq).prio  $\leftarrow$  aux2  $\triangleright$  O(1)
        z  $\leftarrow$  z.izq  $\triangleright$  O(1)
    else
        aux  $\leftarrow$  z.valor  $\triangleright$  O(copiar( $\alpha$ ))
        z.valor  $\leftarrow$  (z.der).valor  $\triangleright$  O(copiar( $\alpha$ ))
        (z.der).valor  $\leftarrow$  aux  $\triangleright$  O(copiar( $\alpha$ ))
        aux2  $\leftarrow$  z.prio  $\triangleright$  O(1)
        z.prio  $\leftarrow$  (z.der).prio  $\triangleright$  O(1)
        (z.der).prio  $\leftarrow$  aux2  $\triangleright$  O(1)
        z  $\leftarrow$  z.der  $\triangleright$  O(1)
    end if
    end while
end function

```

\\ 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

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

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

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 $\equiv \{res =_{obs} vacio()\}$

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 $\equiv \{res =_{obs} claves(d)\}$

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.

7. 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 $\equiv \{\text{true}\}$

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

Complejidad: $O(\log(n) * \text{comparar}(\kappa))$

SIGNIFICADO(**in** d : diccLog(κ, σ), **in** c : κ) $\rightarrow res$: σ

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

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

Complejidad: $O(\log(n) * \text{comparar}(\kappa) + \text{copiar}(\sigma))$

VACIO() $\rightarrow res$: diccLog(κ, σ)

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

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

Complejidad: $O(1)$

DEFINIR(**in/out** d : diccLog(κ, σ), **in** c : κ , **in** s : σ)

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

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

Complejidad: $O(\log(n))$

BORRAR(**in/out** d : diccLog(κ, σ), **in** c : κ)

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

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

Complejidad: $O(\log(n) * \text{comparar}(\kappa) + \max(\text{borrar}(\kappa), \text{borrar}(\sigma)) + \max(\text{copiar}(\kappa), \text{copiar}(\sigma)))$

Representación

diccLog se representa con arb: $argol_{AB}(clave: \kappa, nivel: nat, significado: \sigma)$

// De aquí en adelante por cuestiones de brevedad y legibilidad se referirá al primer elemento del árbol binario (arb.primer) 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 \iff

// 1 $((\forall n_1, n_2 : \text{nodo}(\kappa, \sigma))(n_1 \in \text{arbol}(e) \wedge n_2 \in \text{arbol}(e) \Rightarrow_L (n_1 \in \text{arbol}(n_2.\text{izquierdo}) \Rightarrow n_1.\text{clave} < n_2.\text{clave}) \wedge (n_1 \in \text{arbol}(n_2.\text{derecho}) \Rightarrow n_1.\text{clave} > n_2.\text{clave})) \wedge$

// 2 $((\forall n : \text{nodo}(\kappa, \sigma))(n \in \text{arbol}(e) \wedge n.\text{izquierdo} = \text{NULO} \wedge n.\text{derecho} = \text{NULO} \Rightarrow n.\text{nivel} = 1) \wedge$

// 3 $((\forall n : \text{nodo}(\kappa, \sigma))(n \in \text{arbol}(e) \wedge n.\text{izquierdo} \neq \text{NULO} \Rightarrow (n.\text{izquierdo}).\text{nivel} = n.\text{nivel} - 1) \wedge$

// 4 $((\forall n : \text{nodo}(\kappa, \sigma))(n \in \text{arbol}(e) \wedge n.\text{derecho} \neq \text{NULO} \Rightarrow ((n.\text{derecho}).\text{nivel} = n.\text{nivel} - 1 \vee (n.\text{derecho}).\text{nivel} = n.\text{nivel})) \wedge$

// 5 $((\forall n : \text{nodo}(\kappa, \sigma))(n \in \text{arbol}(e) \wedge n.\text{derecho} \neq \text{NULO} \wedge_L (n.\text{derecho}).\text{derecho} \neq \text{NULO} \Rightarrow_L ((n.\text{derecho}).\text{derecho}).\text{nivel} < n.\text{nivel})) \wedge$

// 6 $((\forall n : \text{nodo}(\kappa, \sigma))(n \in \text{arbol}(e) \wedge n.\text{nivel} > 1 \Rightarrow (n.\text{izquierdo} \neq \text{NULO} \wedge n.\text{derecho} \neq \text{NULO}))$

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

Abs(d) \equiv c: $\text{dicc}(\kappa, \sigma) \mid ((\forall k : \kappa)(k \in \text{claves}(c) \Rightarrow (\exists n : \text{estr}(\kappa, \sigma))(n \in \text{arbol}(d) \wedge n.\text{clave} = k)) \wedge ((\forall n : \text{nodo}(\kappa, \sigma))(n \in \text{arbol}(d) \Rightarrow n.\text{clave} \in \text{claves}(c)))) \wedge_L$

$(\forall n : \text{nodo}(\kappa, \sigma))(n \in \text{arbol}(d) \Rightarrow \text{obtener}(c, n.\text{clave}) =_{\text{obs}} n.\text{significado})$

Especificación de las operaciones auxiliares utilizadas para Rep y Abs

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

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

Algoritmos

Algorithm 36 Implementación de Definido?

```

function IDEFINIDO?(in d: estr , in c:  $\kappa$ )  $\rightarrow$  res: bool
  \ \ Búsqueda estándar en un ABB
  nodoActual  $\leftarrow$  d  $\triangleright O(1)$ 
  res  $\leftarrow$  FALSE  $\triangleright O(1)$ 
  while  $\neg$ nil?(nodoActual) &&  $\neg$ res do  $\triangleright$  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  $\pi_1(\text{raiz}(\text{nodoActual})) == c$  then  $\triangleright O(\text{comparar}(\kappa))$ 
      res  $\leftarrow$  TRUE  $\triangleright O(1)$ 
    else
      if  $c < \pi_1(\text{raiz}(\text{nodoActual}))$  then  $\triangleright O(\text{comparar}(\kappa))$ 
        nodoActual  $\leftarrow$  izq(nodoActual)  $\triangleright O(1)$ 
      else
        nodoActual  $\leftarrow$  der(nodoActual)  $\triangleright O(1)$ 
      end if
    end if
  end while
end function

```

Algorithm 37 Implementación de Significado

```

function ISIGNIFICADO(in d: estr , in c:  $\kappa$ )  $\rightarrow$  res:  $\sigma$ 
  \ \ Búsqueda estándar en un ABB
  nodoActual  $\leftarrow$  d  $\triangleright O(1)$ 
  while  $\neg$ nil?(nodoActual) &&  $\neg$ res do  $\triangleright$  El ciclo se ejecuta en el peor caso  $O(\log(n))$  veces.
    if  $\pi_1(\text{raiz}(\text{nodoActual})) == c$  then  $\triangleright O(\text{comparar}(\kappa))$ 
      res  $\leftarrow$   $\pi_3(\text{raiz}(\text{nodoActual}))$   $\triangleright O(\text{copiar}(\sigma))$ . Esta operacion solo se ejecuta una vez (implica
       $\neg$ guarda del ciclo que la contiene).
    else
      if  $c < \pi_1(\text{raiz}(\text{nodoActual}))$  then  $\triangleright O(\text{comparar}(\kappa))$ 
        nodoActual  $\leftarrow$  izq(nodoActual)  $\triangleright O(1)$ 
      else
        nodoActual  $\leftarrow$  der(nodoActual)  $\triangleright O(1)$ 
      end if
    end if
  end while
end function

```

Algorithm 38 Implementación de Vacio

```

function iVACIO  $\rightarrow$  res: estr
    res  $\leftarrow$  nil()  $\triangleright O(1)$ 
end function

```

Algorithm 39 Implementación de Definir

```

function iDEFINIR(inout d: estr, in c:  $\kappa$ , in s:  $\sigma$ )
    // Si ya se llego a una hoja, se inserta el nuevo elemento
    if nil?(d) then  $\triangleright O(1)$ 
        res  $\leftarrow$   $\langle c, NULO, NULO, s, 1 \rangle$   $\triangleright O(\max(\text{copiar}(\kappa), \text{copiar}(\sigma)))$ 
    // Se busca la posicion correspondiente al nuevo nodo (antes de rebalancear el arbol).
    else if c  $< \pi_1(\text{raiz}(d))$  then  $\triangleright O(\text{comparar}(\kappa))$ 
        setearIzq(d, iDefinir(izquierdo(d), c, s))  $\triangleright$  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  $> \pi_1(\text{raiz}(d))$  then  $\triangleright O(\text{comparar}(\kappa))$ 
        setearDer(d, iDefinir(d.derecho, c, s))  $\triangleright$  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  $\leftarrow$  Torsion(d)  $\triangleright O(1)$ 
    d  $\leftarrow$  Division(d)  $\triangleright O(1)$ 
end function

```

Algorithm 40 Implementación de Torsion

```

function iTORSION(in d: estr)  $\rightarrow$  res: estr
    // Si el nodo tiene un hijo izquierdo del mismo nivel se debe realizar una rotacion para restaurar el invariante.
    if nil?(d)  $\parallel$  nil?(izq(d)) then  $\triangleright O(1)$ 
        res  $\leftarrow$  d  $\triangleright O(1)$ 
    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  $\pi_2(\text{raiz}(\text{izq}(d))) = \pi_2(d)$  then  $\triangleright O(1)$ 
            nodoAux  $\leftarrow$  izq(d)  $\triangleright O(1)$ 
            setearIzq(d, der(nodoAux))  $\triangleright O(1)$ 
            setearDer(nodoAux, d)  $\triangleright O(1)$ 
            res  $\leftarrow$  nodoAux  $\triangleright O(1)$ 
        else
            res  $\leftarrow$  d  $\triangleright O(1)$ 
        end if
    end if
end function

```

Algorithm 41 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)
      setearVal(nodoAux, < $\pi_1$ (raiz(nodoAux)),  $\pi_2$ (raiz(nodoAux))++,  $\pi_3$ (raiz(nodoAux))>) ▷ O(1)
      res ← nodoAux                                                         ▷ O(1)
    else
      res ← d                                                                ▷ O(1)
    end if
  end if
end function

```

Algorithm 42 Implementación de Borrar

```

function IBORRAR(inout d: estr, in c:  $\kappa$ )
  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( $\kappa$ ))
    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( $\kappa$ ))
      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)
  \ -----VER ESTA LINEAAAAA-----
    borrar(d) ▷ O(max(borrar( $\kappa$ ), borrar( $\sigma$ )))
    d  $\leftarrow$  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  $\leftarrow$  der(d) ▷ O(1)
    while  $\neg$ nil?(izq(aux)) do ▷ En el peor caso el ciclo se ejecuta O(log(n)) veces.
      aux  $\leftarrow$  izq(aux) ▷ O(1)
    end while
  \ Se hace un swap y se elimina el elemento.
    setearDer(d, iBorrar( $\pi_1$ (raiz(aux)), der(d)))
    setearVal(d, < $\pi_1$ (raiz(aux)),  $\pi_2$ (raiz(d)),  $\pi_3$ (raiz(d))>) ▷ O(copiar( $\kappa$ ))
    setearVal(d, < $\pi_1$ (raiz(d)),  $\pi_2$ (raiz(d)),  $\pi_3$ (raiz(aux))>) ▷ O(copiar( $\sigma$ ))
  else
  \ Se busca el predecesor del elemento (bajando una vez por la rama derecha y luego por la izquierda hasta
  encontrar una hoja).
    aux  $\leftarrow$  izq(d) ▷ O(1)
    while  $\neg$ nil?(der(aux)) do ▷ En el peor caso el ciclo se ejecuta O(log(n)) veces.
      aux  $\leftarrow$  der(aux) ▷ O(1)
    end while
  \ Se hace un swap y se elimina el elemento.
    setearIzq(d, iBorrar( $\pi_1$ (raiz(aux)), izq(d)))
    setearVal(d, < $\pi_1$ (raiz(aux)),  $\pi_2$ (raiz(d)),  $\pi_3$ (raiz(d))>) ▷ O(copiar( $\kappa$ ))
    setearVal(d, < $\pi_1$ (raiz(d)),  $\pi_2$ (raiz(d)),  $\pi_3$ (raiz(aux))>) ▷ O(copiar( $\sigma$ ))
  end if
  \ Se nivela, divide y tuerce para restaurar el invariante.
  d  $\leftarrow$  Nivelar(T) ▷ O(1)
  d  $\leftarrow$  Torsion(T) ▷ O(1)
  if  $\neg$ nil?(der(d)) then ▷ O(1)
    setearDer(der(d), Torsion(der(der(d)))) ▷ O(1)
  end if
  d  $\leftarrow$  Division(T) ▷ O(1)
  setearDer(d, Division(der(d))) ▷ O(1)
  res  $\leftarrow$  d ▷ O(1)
end function
procedure NIVELAR(inout d: estr)
  nivel_correcto  $\leftarrow$  min( $\pi_2$ (raiz(izq(d))),  $\pi_2$ (raiz(der(d))))+1 ▷ O(1)
  if nivel_correcto <  $\pi_2$ (raiz(d)) then ▷ O(1)
    setearVal(d, < $\pi_1$ (raiz(d)), nivel_correcto,  $\pi_3$ (raiz(d))>) ▷ O(1)
    if nivel_correcto <  $\pi_2$ (raiz(der(d))) then ▷ O(1)
      setearVal(der(d), < $\pi_1$ (raiz(der(d))), nivel_correcto,  $\pi_3$ (raiz(der(d))))>) ▷ O(1)
    end if
  end if
end procedure

```
