

# 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	15
3. Módulo Cola de Prioridad $_{HEAP} (\alpha)$	26
4. Módulo Diccionario $_{TRIE} (\alpha)$	32
5. Módulo Diccionario $_{AA} (\kappa, \sigma)$	33

# 1. Módulo Red

## Interfaz

**usa:** CONJ( $\alpha$ ), ITCONJ( $\alpha$ ), LISTA( $\alpha$ ), ITLISTA( $\alpha$ ).

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

**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.

USAIINTERFAZ?(in  $r : \text{red}$ , in  $c : \text{hostname}$ , 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.

CAMINOSMINIMOS(in  $r : \text{red}$ , in  $c_1 : \text{hostname}$ , 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.

HAYCAMINO?(in  $r : \text{red}$ , in  $c_1 : \text{hostname}$ , 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.

$\bullet == \bullet$  (in  $r_1 : \text{red}$ , 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.

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

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

**Post**  $\equiv \{res =_{obs} r\}$

**Descripción:** copia la red.

**Aliasing:** res se devuelve por copia

**donde:**

hostname es string,

interfaz es nat,

compu es tupla<ip: hostname, interfaces: conj(interfaz)>.

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

#### TAD RED EXTENDIDA

**extiende** RED

**otras operaciones**

damehostnames : conj(compu)  $\rightarrow$  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.

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

\\ 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)

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

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

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

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( $e$ )  $\equiv r$  | computadoras( $r$ ) = 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**      $\forall e: \text{dicc}(\text{hostname}, X), \forall d: \text{dicc}(\text{interfaz}, \text{hostname}), \forall cc: \text{conj}(\text{hostname}), \forall ci: \text{conj}(\text{interfaz}), \forall h: \text{hostname}$

$$\text{dameComputadoras}(e) \equiv \text{auxDameComputadoras}(e, \text{claves}(e))$$

```

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

```

$$\text{buscarClave}(d, h) \equiv \text{auxBuscarClave}(d, h, \text{claves}(d))$$

```

auxBuscarClave( $d, h, ci$ )  $\equiv$  if obtener( $d, \text{dameUno}(cc)$ ) =  $h$  then
    dameUno( $cc$ )
else
    auxBuscarClave( $d, h, \text{sinUno}(ci)$ )
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)
    DefinirRapido(Significado(r, c2).conexiones, i2, c1)
    // Actualizo caminos de ambas
    ActualizarCaminos(r, c1, c2)
    ActualizarCaminos(r, c2, c1)
    // Creo conjunto con los actualizados hasta el momento
    actualizados ← Vacio()                        ▷ conjunto
    AgregarRapido(actualizados, c1)
    AgregarRapido(actualizados, c2)
    // Actualizo caminos del resto de la Red por recursion
    ActualizarVecinos(r, c1, actualizados)
end function

```

---

**Algorithm 7** Implementación de función auxiliar Actualizar Caminos

---

```

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

```

---

**Algorithm 8** Implementación de función auxiliar Actualizar Vecinos

---

```

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

```

---

**Algorithm 9** Implementación de Vecinos

---

```

function IVECINOS(inout r: estr_red, in c1: hostname)→ res: conj(hostname)
  it ← CrearIt(Significado(r,c1).conexiones)                                ▷ O(1) + O(n)
  res ← Vacio()                                                            ▷ 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()                                                            ▷ 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 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 ¬(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( $\alpha$ ), ITCONJ( $\alpha$ ), LISTA( $\alpha$ ), ITLISTA( $\alpha$ ), DICC<sub>UNIV</sub>( $\kappa, \sigma$ ), DICC<sub>LOG</sub>( $\kappa, \sigma$ ), COLA<sub>LOG</sub>( $\alpha$ ).

**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()

**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()

**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()

**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()

**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()

**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()

**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:**  $O()$

**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:**  $O()$

**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:**  $O()$

**Descripción:** devuelve la computadora que más paquetes envió.

**Aliasing:** res se devuelve por copia.

$\bullet = \bullet(\text{in } d_1 : \text{dcnet}, \text{in } d_2 : \text{dcnet}) \rightarrow res : \text{bool}$

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

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

**Complejidad:**  $O()$

**Descripción:** indica si dos dcnet son iguales.

donde:

hostname es string,

interfaz es nat,

IDpaquete es nat,

compu es tupla<ip: hostname, interfaces: conj(interfaz)>,

paquete es tupla<id: IDpaquete, prioridad: nat, origen: hostname, destino: hostname >.



**Especificación de las operaciones auxiliares utilizadas en la interfaz (no exportadas)****TAD DCNET EXTENDIDA****extiende** DCNET**otras operaciones**

damehostnames : conj(compu)  $\rightarrow$  conj(hostname)  
dameCompu : dcnet  $d \times$  hostname  $s \rightarrow$  compu  
 $\{s \in \text{dameHostnames}(\text{computadoras}(\text{red}(d)))\}$   
auxDameCompu : hostname  $s \times$  conj(compu)  $cc \rightarrow$  compu  
dameSecuDeHostnames : secu(compu)  $\rightarrow$  secu(hostname)  
IDpaqueteEnTransito? : dcnet  $d \times$  IDpaquete  $p \rightarrow$  bool  
damePaquete : dcnet  $d \times$  IDpaquete  $p \rightarrow$  paquete  
 $\{\text{IDpaqueteEnTransito?}(d,p)\}$   
dameIDpaquetes : conj(paquete)  $\rightarrow$  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 )

```

*\\ 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 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 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 Alaternativa adjunto).
- Para todos los paquetes de una computadora:
- 12: El origen y el destino estan entre las computadoras de la dcnet.
  - 13: El origen y el destino son distintos.
  - 14: Hay un camino posible entre el origen y el destino.
  - 15: La computadora actual esta en el camino minimo entre el origen y el destino.
  - 16: El id es unico.
  - 17: Son accesibles por el dicc usando su ID.
  - 18: En la cola hay un iterador a cada paquete.

Rep : estr\_dcnet  $\longrightarrow$  bool

Rep( $e$ )  $\equiv$  true  $\iff$

```

\\ 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  $0 < \text{obtener}(\text{computadoras}, c).\text{indice} < \#\text{claves}(\text{computadoras})-1 \wedge$ 
\\ 5  $\neg(\exists c': \text{hostname}, c' \in \text{claves}(\text{computadoras}), c \neq c') \text{obtener}(\text{computadoras}, c').\text{indice}$ 
=  $\text{obtener}(\text{computadoras}, c).\text{indice} \wedge$ 
\\ 6  $\neg(\exists c': \text{hostname}, c' \in \text{claves}(\text{computadoras}), c \neq c') \text{obtener}(\text{computadoras}, c').\text{cantEnvios} > \text{SiguienteSignificado}(e.\text{conMasEnvios}).\text{cantEnvios} \wedge$ 
\\ 7  $\text{tam}(e.\text{caminos}) = \#\text{claves}(e.\text{computadoras}) \wedge_L (\forall i: \text{nat}, 0 < i < \#\text{claves}(e.\text{computadoras})-1)$ 
 $\text{tam}(e.\text{caminos}[i]) = \#\text{claves}(e.\text{computadoras}) \wedge$ 
\\ 8 ( $\forall c1, c2$ : hostname,  $c1, c2 \in$  claves( $e$ .porHostname))
 $\neg \emptyset? (\text{caminosMinimos}(e.\text{red}, \text{dameCompu}(c1), \text{dameCompu}(c2))) \Rightarrow$ 
 $e.\text{caminos}[\text{obtener}(e.\text{computadoras}, c1).\text{indice}][\text{obtener}(e.\text{computadoras}, c2).\text{indice}] =$ 
 $\text{dameUno}(\text{caminosMinimos}(e.\text{red}, \text{dameCompu}(c1), \text{dameCompu}(c2))) \wedge$ 
 $\emptyset? (\text{caminosMinimos}(e.\text{red}, \text{dameCompu}(c1), \text{dameCompu}(c2))) \Rightarrow$ 
 $e.\text{caminos}[\text{obtener}(e.\text{computadoras}, c1).\text{indice}][\text{obtener}(e.\text{computadoras}, c2).\text{indice}] = \text{Vacía}() \wedge$ 
( $\forall c$ : hostname,  $c \in$  claves( $e$ .computadoras)) (
\\ 9  $\text{dameIDpaquetes}(\text{obtener}(e.\text{computadoras}, c).\text{paquetes}) = \text{claves}(\text{obtener}(e.\text{computadoras}, c).\text{paquetesPorID}) \wedge$ 
\\ 10  $\#(\text{obtener}(e.\text{computadoras}, c).\text{paquetes}) = \#(\text{obtener}(e.\text{computadoras}, c).\text{cola}) \wedge$ 
\\ 11  $\text{vacía}?( \text{obtener}(e.\text{computadoras}, c).\text{cola}) = \emptyset?(\text{obtener}(e.\text{computadoras}, c).\text{paquetes}) \wedge$ 
 $\text{Siguiente}(\Pi_2(\text{proximo}(\text{obtener}(e.\text{computadoras}, c).\text{cola}))) \in \text{obtener}(e.\text{computadoras}, c).\text{paquetes}$ 
 $\wedge \neg(\exists p': \text{paquete}, p' \in \text{obtener}(e.\text{computadoras}, c).\text{paquetes}) p'.\text{prioridad} <$ 
 $\text{Siguiente}(\Pi_2(\text{proximo}(\text{obtener}(e.\text{computadoras}, c).\text{cola}))).\text{prioridad} \wedge$ 
 $\Pi_1(\text{proximo}(\text{obtener}(e.\text{computadoras}, c).\text{cola})) = \text{Siguiente}(\Pi_2(\text{proximo}(\text{obtener}(e.\text{computadoras}, c).\text{cola}))).\text{prioridad} \wedge$ 
 $\text{desencolar}(\text{obtener}(e.\text{computadoras}, c).\text{cola}) = \text{armarCola}(\text{obtener}(e.\text{computadoras}, c).\text{paquetes} - \{\text{Siguiente}(\Pi_2(\text{proximo}(\text{obtener}(e.\text{computadoras}, c).\text{cola})))\}) \wedge$ 
( $\forall p$ : paquete,  $p \in \text{obtener}(e.\text{computadoras}, c).\text{paquetes}$ ) (
\\ 12  $\text{origen}(p).\text{ip} \in \text{claves}(e.\text{computadoras}) \wedge \text{destino}(p).\text{ip} \in \text{claves}(e.\text{computadoras}) \wedge$ 
\\ 13  $\text{origen}(p).\text{ip} \neq \text{destino}(p).\text{ip} \wedge$ 
\\ 14  $\text{hayCamino?}(e.\text{red}, \text{origen}(p), \text{destino}(p)) \wedge$ 
\\ 15  $\text{está?}(c, \text{caminos}[\text{obtener}(e.\text{computadoras}, \text{origen}(p).\text{ip})][\text{obtener}(e.\text{computadoras}, \text{destino}(p).\text{ip})]) \wedge$ 
\\ 16 ( $\forall c': \text{hostname}, c' \in \text{claves}(e.\text{computadoras}), c' \neq c$ )  $\neg(\exists p': \text{paquete}, p' \in \text{obtener}(e.\text{computadoras}, c').\text{paquetes}, p \neq p') p.\text{id} = p'.\text{id}$ 
\\ 17  $\text{definido?}(\text{obtener}(e.\text{computadoras}, c).\text{paquetesPorID}, p.\text{id}) \wedge_L$ 
 $\text{Siguiente}(\text{obtener}(\text{obtener}(e.\text{computadoras}, c).\text{paquetesPorID}, p.\text{id})) = p \wedge$ 
\\ 18 ( $\exists it$ : itConj(paquete),  $it \in \text{obtener}(e.\text{computadoras}, c).\text{cola}$ )  $\text{Siguiente}(it) = p$  ) )
```

### Especificación de las funciones auxiliares utilizadas en Rep

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

**axiomas**  $\forall cc$ : conj(paquete)

```

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

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

## Algoritmos

---

### Algorithm 15 Implementación de Red

---

```

function IRED(in  $d: \text{estr\_dcnet}$ )  $\rightarrow$   $\text{res: Red}$ 
   $\text{res} \leftarrow d.\text{red}$ 
end function

```

---



---

### Algorithm 16 Implementación de CaminoRecorrido

---

```

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

```

---



---

### Algorithm 17 Implementación de paquetes enviados

---

```

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

```

---

**Algorithm 18** Implementación EnEspera

---

```

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

```

---

**Algorithm 19** Implementación de iniciarDCNet

---

```

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

```

---

---

**Algorithm 20** Implementación de crearPaquete

---

```
function ICREARPAQUETE(in/out  $d$ : estr_dcnet, in  $p$ : paquete)
    itPC  $\leftarrow$  Significado( $d$ .porHostname, paquete.origen)  $\triangleright O(L)$ 
    itPaq  $\leftarrow$  AgregarRapido(SiguienteSignificado(itPC).paquetes,  $p$ )  $\triangleright O(\text{copy}(p))$ 
    Encolar(SiguienteSignificado(itPC).cola,  $p$ .prioridad, itPaq)  $\triangleright O(\log(n))$ ,  $n$  cantidad de nodos
    Definir(SiguienteSignificado(itPC).paquetesPorID, IDpaquete, itPaq)  $\triangleright O(\log(n))$ ,  $n$  cantidad de
    nodos
end function
```

---

**Algorithm 21** Implementación de AvanzarSegundo

---

```

function IAVANZARSEGUNDO(inout d: estr_dcnet)
    arreglo ← crearArreglo[#Claves(d.computadoras)] de tupla(usedo: 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].usedo = 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

```

---



---

**Algorithm 22** Implementación de PaqueteEnTransito?

---

```

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

```

---



---

**Algorithm 23** Implementación de LaQueMasEnvio

---

```

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

```

---



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

```

---

### 3. Módulo Cola de Prioridad $_{HEAP}(\alpha)$

#### Interfaz

**usa:** TUPLA, NAT, BOOL,  $\alpha$ .

**se explica con:** COLA DE PRIORIDAD ALTERNATIVA.

**géneros:** colaLog( $\alpha$ ).

#### Operaciones de Cola de Prioridad $_{HEAP}$

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

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

**Post**  $\equiv \{res =_{obs} \text{vacía}\}$

**Complejidad:**  $O(1)$

**Descripción:** Crea una cola vacía.

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

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

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

**Complejidad:**  $O(1)$

**Descripción:** Indica si la cola está vacía.

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

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

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

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

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

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

**Pre**  $\equiv \{estr = estr_0\}$

**Post**  $\equiv \{res \wedge estr =_{obs} \text{encolar}(estr_0)\}$

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

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

DESENCOLAR(in/out  $estr : \text{colaLog}(\alpha)$ )  $\rightarrow res : \alpha$

**Pre**  $\equiv \{estr = estr_0 \wedge \neg \text{Vacía?}(estr)\}$

**Post**  $\equiv \{estr =_{obs} \text{desencolar}(estr_0) \wedge res =_{obs} \text{proximo}(estr_0)\}$

**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**( $\alpha$ )**géneros**      colaPrio( $\alpha$ )**exporta**      colaPrio( $\alpha$ ), generadores, observadores**usa**            BOOL, NAT, TUPLA**observadores básicos**vacía?        : colaPrior( $\alpha$ )                     $\longrightarrow$  boolpróximo     : colaPrior( $\alpha$ )  $c$                      $\longrightarrow$  tupla( $nat, \alpha$ )                     $\{ \neg \text{vacía?}(c) \}$ desencolar : colaPrior( $\alpha$ )  $c$                      $\longrightarrow$  colaPrior( $\alpha$ )                     $\{ \neg \text{vacía?}(c) \}$ **generadores**vacía        :     $\longrightarrow$  colaPrior( $\alpha$ )encolar     :  $nat \times \alpha \times \text{colaPrior}(\alpha)$     $\longrightarrow$  colaPrior( $\alpha$ )**axiomas**       $\forall c: \text{colaPrior}(\alpha), \forall e: \alpha$ vacía?(vacía)                     $\equiv$  truevacía?(encolar( $p, e, c$ ))        $\equiv$  falsepróximo(encolar( $p, e, c$ ))       $\equiv$  **if** vacía?( $c$ )  $\vee_L \Pi_1(\text{proximo}(c)) < p$  **then**  $< p, e >$  **else if**  
 $\Pi_1(\text{proximo}(c)) = p$  **then**  $< p, e > \vee \text{proximo}(c)$  **else proximo}(c) **fi**  
**fi****desencolar(encolar( $p, e, c$ ))  $\equiv$  **if** vacía?( $c$ )  $\vee_L \Pi_1(\text{proximo}(c)) < p$  **then**  $c$  **else if**  
 $\Pi_1(\text{proximo}(c)) = p$  **then**  $c \vee \text{encolar}(p, e, \text{desencolar}(c))$  **el-**  
**se encolar}(p, e, \text{desencolar}(c)) **fi fi******Fin TAD**

## Representación

$\text{colaLog}(\alpha)$  se representa con  $\text{estr\_heap}(\alpha)$

donde  $\text{estr\_heap}(\alpha)$  es  $\text{tupla}(\text{size: nat}$   
 $\text{primero: nodo}(\alpha)$   
 $)$

donde  $\text{nodo}(\alpha)$  es  $\text{tupla}(\text{padre: puntero}(\text{nodo}(\alpha))$   
 $\text{izq: puntero}(\text{nodo}(\alpha))$   
 $\text{der: puntero}(\text{nodo}(\alpha))$   
 $\text{prio: nat}$   
 $\text{valor: } \alpha$   
 $)$

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

$\text{Abs: estr\_heap}(\alpha) \rightarrow \text{colaPrio}(\alpha) \{ \text{Rep}(e) \}$   
 $\text{Abs}(e) \equiv c : \text{colaPrio}(\alpha) \mid \text{Vacía?}(e) = \text{Vacía?}(c) \wedge_L$   
 $(\neg \text{Vacía?}(e) \Rightarrow \text{Proximo}(e) = \text{Proximo}(c)) \wedge$   
 $(\neg \text{Vacía?}(e) \Rightarrow \text{Desencolar}(e) = \text{Desencolar}(c))$

### Especificación de las operaciones auxiliares utilizadas para Rep y Abs

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

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

$\text{arbol}(n) \equiv \text{if } n.\text{izq} \neq \text{null} \wedge n.\text{der} \neq \text{null} \text{ then}$   
 $\text{Ag}(n.\text{valor}, \text{arbol}(n.\text{izq}) \cup \text{arbol}(n.\text{der}))$   
 $\text{else}$   
 $\text{if } n.\text{izq} \neq \text{null} \text{ then}$   
 $\text{Ag}(n.\text{valor}, \text{arbol}(n.\text{izq}))$   
 $\text{else}$   
 $\text{if } n.\text{der} \neq \text{null} \text{ then } \text{Ag}(n.\text{valor}, \text{arbol}(n.\text{der})) \text{ else } \text{Ag}(n.\text{valor}, \emptyset) \text{ fi}$   
 $\text{fi}$   
 $\text{caminoHastaRaiz}(n) \equiv \text{if } n.\text{padre} = \text{null} \text{ then } 0 \text{ else } \text{caminoHastaRaiz}(n.\text{padre}) + 1 \text{ fi}$

## Algoritmos

---

**Algorithm 25** Implementación de Vacía

---

```
function iVACIA  $\rightarrow res$ : colaLog( $\alpha$ )  
     $res \leftarrow \langle 0, null \rangle$   $\triangleright O(1)$   
end function
```

---

---

**Algorithm 26** Implementación de Vacía?

---

```
function iVACIA?(in  $estr$ : estr_heap( $\alpha$ ))  $\rightarrow res$ : bool  
     $res \leftarrow (estr.primerero == null)$   $\triangleright O(1)$   
end function
```

---

---

**Algorithm 27** Implementación de Próximo

---

```
function PRÓXIMO(in  $estr$ : estr_heap( $\alpha$ ))  $\rightarrow res$ : tupla( $nat, \alpha$ )  
     $res \leftarrow \langle (estr.primerero).prioridad, (estr.primerero).valor \rangle$   $\triangleright O(\text{copiar}(\alpha))$   
end function
```

---

**Algorithm 28** 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 estr.size == 0 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 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 $\rangle$   $\triangleright O(\text{copiar}(\alpha))$ 
        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 29** 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  $\rangle$   $\triangleright O(\text{copiar}(\alpha))$ 
    w.padre  $\leftarrow$  z  $\triangleright O(1)$ 
    if prim(y) == 0 then  $\triangleright O(1)$ 
        z.izq  $\leftarrow$  puntero(w)  $\triangleright O(1)$ 
    else  $\triangleright O(1)$ 
        z.der  $\leftarrow$  puntero(w)  $\triangleright O(1)$ 
    end if
    (estr.primer).valor  $\leftarrow$  z.valor  $\triangleright O(\text{copiar}(\alpha))$ 
    (estr.primer).prio  $\leftarrow$  z.prio  $\triangleright O(1)$ 
    borrar(z)  $\triangleright O(\text{borrar}(\alpha))$ 
    z  $\leftarrow$  estr.primer  $\triangleright O(1)$ 
    size  $\leftarrow$  —  $\triangleright O(1)$ 
    while (z.izq  $\neq$  null  $\vee$  z.der  $\neq$  null)  $\wedge_L$  z.prio < maxPrio(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  $O(\log(n))$ 
         $\backslash \backslash$  maxPrio devuelve la maxima prioridad si ambos punteros son validos, o la prioridad apuntada por el
        puntero no nulo en caso de que alguno no lo sea
        if z.der == null  $\vee_L$  (z.izq).prio  $\geq$  (z.der).prio then  $\triangleright O(1)$ 
            aux  $\leftarrow$  z.valor  $\triangleright O(\text{copiar}(\alpha))$ 
            z.valor  $\leftarrow$  (z.izq).valor  $\triangleright O(\text{copiar}(\alpha))$ 
            (z.izq).valor  $\leftarrow$  aux  $\triangleright O(\text{copiar}(\alpha))$ 
            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  $\triangleright O(1)$ 
            aux  $\leftarrow$  z.valor  $\triangleright O(\text{copiar}(\alpha))$ 
            z.valor  $\leftarrow$  (z.der).valor  $\triangleright O(\text{copiar}(\alpha))$ 
            (z.der).valor  $\leftarrow$  aux  $\triangleright O(\text{copiar}(\alpha))$ 
            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

```

---

## 4. Módulo Diccionario *TRIE* ( $\alpha$ )

### Interfaz

se explica con: `DICCIONARIO(String,  $\sigma$ )`.

géneros: `diccUniv( $\kappa, \sigma$ )`.

### Operaciones

**DEFINIDA**(in  $d$ : `diccUniv(String,  $\sigma$ )`, 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,  $\sigma$ )`, in  $c$ : `String`)  $\rightarrow res$  :  $\sigma$

**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,  $\sigma$ )`

**Pre**  $\equiv \{true\}$

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

**Complejidad**:  $O(1)$

**Descripción**: Crea un diccionario vacío.

**DEFINIR**(in/out  $d$ : `diccUniv(String,  $\sigma$ )`, in  $c$ : `String`, in  $s$ :  $\sigma$ )  $\rightarrow res$  : `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,  $\sigma$ )`, 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,  $\sigma$ )`)  $\rightarrow res$  : `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.



## 5. Módulo Diccionario $_{AA}(\kappa, \sigma)$

### Interfaz

**usa:** .

**se explica con:**  $\text{DICCIONARIO}(\kappa, \sigma)$ .

**géneros:**  $\text{diccLog}(\kappa, \sigma)$ .

### Operaciones

**DEFINIDO?**(**in**  $d: \text{diccLog}(\kappa, \sigma)$ , **in**  $c: \kappa$ )  $\rightarrow res: \text{Bool}$

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

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

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

**SIGNIFICADO**(**in**  $d: \text{diccLog}(\kappa, \sigma)$ , **in**  $c: \kappa$ )  $\rightarrow res: \sigma$

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

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

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

**VACIO**()  $\rightarrow res: \text{diccLog}(\kappa, \sigma)$

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

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

**Complejidad:**  $O(1)$

**DEFINIR**(**in/out**  $d: \text{diccLog}(\kappa, \sigma)$ , **in**  $c: \kappa$ , **in**  $s: \sigma$ )

**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**  $d: \text{diccLog}(\kappa, \sigma)$ , **in**  $c: \kappa$ )

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

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

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

## Representación

diccLog se representa con puntero( $\text{estr}(\kappa, \sigma)$ )

donde  $\text{estr}(\kappa, \sigma)$  es  $\text{tupla}(\text{clave: } \kappa$   
 $\text{izquierdo: puntero}(\text{nodo}(\kappa, \sigma))$   
 $\text{derecho: puntero}(\text{nodo}(\kappa, \sigma))$   
 $\text{significado: } \sigma$   
 $\text{nivel: nat } )$

Rep:  $\text{estr} \rightarrow \text{bool}$

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

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

$\text{Abs}(d) \equiv c: \text{dicc}(\kappa, \sigma) \mid ((\forall k : \kappa) (k \in \text{claves}(c) \Rightarrow \text{Definido?}(c, k)) \wedge$   
 $(\text{Definido?}(c, k) \Rightarrow k \in \text{claves}(c))) \wedge_L$   
 $(\forall k : \kappa) (k \in \text{claves}(c) \Rightarrow_L \text{obtener}(c, k) =_{\text{obs}} \text{Significado}(d, k))$

### Especificación de las operaciones auxiliares utilizadas para Rep y Abs

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

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

## Algoritmos

---

**Algorithm 30** Implementación de Definido?

---

```
function IDEFINIDO?(in d: estr , in c:  $\kappa$ )  $\rightarrow$  res: bool
  nodoActual  $\leftarrow$  d
  res  $\leftarrow$  FALSE
  while  $\neg$ (nodoActual == NULO) &&  $\neg$ res do
    if nodoActual.clave == c then
      res  $\leftarrow$  TRUE
    else
      if c < nodoActual.clave then
        nodoActual  $\leftarrow$  nodoActual.izquierdo
      else
        nodoActual  $\leftarrow$  nodoActual.derecho
      end if
    end if
  end while
end function
```

---

---

**Algorithm 31** Implementación de Significado

---

```
function ISIGNIFICADO(in d: estr , in c:  $\kappa$ )  $\rightarrow$  res:  $\sigma$ 
  nodoActual  $\leftarrow$  d
  while  $\neg$ (nodoActual == NULO) &&  $\neg$ res do
    if nodoActual.clave == c then
      res  $\leftarrow$  nodoActual.significado
    else
      if c < nodoActual.clave then
        nodoActual  $\leftarrow$  nodoActual.izquierdo
      else
        nodoActual  $\leftarrow$  nodoActual.derecho
      end if
    end if
  end while
end function
```

---

---

**Algorithm 32** Implementación de Vacio

---

```
function IVACIO  $\rightarrow$  res: estr
  res  $\leftarrow$  NULO
end function
```

---

---

**Algorithm 33** Implementación de Definir

---

```

function IDEFINIR(inout d: estr , in c:  $\kappa$ , in s:  $\sigma$ )
  if d == NULO then
     $res \leftarrow \langle c, NULO, NULO, s, 1 \rangle$ 
  else if  $c < d.clave$  then
     $d.izquierdo \leftarrow iDefinir(d.izquierdo, c, s)$ 
  else if  $c > d.clave$  then
     $d.derecho \leftarrow iDefinir(d.derecho, c, s)$ 
  end if
   $d \leftarrow Torsion(d)$ 
   $d \leftarrow Division(d)$ 
   $res \leftarrow d$ 
end function

```

---



---

**Algorithm 34** Implementación de Torsion

---

```

function ITORSION(in d: estr)  $\rightarrow$  res: estr
  if d == NULO  $\parallel$  d.izquierdo == NULO then
     $res \leftarrow d$ 
  else
    if (d.izquierdo).nivel  $\geq$  d.nivel then
      nodoAux  $\leftarrow$  d.izquierdo
      d.izquierdo  $\leftarrow$  nodoAux.derecho
      nodo.derecho  $\leftarrow$  d
       $res \leftarrow$  nodoAux
    else
       $res \leftarrow d$ 
    end if
  end if
end function

```

---



---

**Algorithm 35** Implementación de Division

---

```

function IDIVISION(in d: estr)  $\rightarrow$  res: estr
  if d == NULO  $\parallel$  d.derecho == NULO  $\parallel$  d.derecho.derecho == NULO then
     $res \leftarrow d$ 
  else
    if (d.derecho.derecho).nivel == d.nivel then
      nodoAux  $\leftarrow$  d.derecho
      d.derecho  $\leftarrow$  nodoAux.izquierdo
      nodoAux.izquierdo  $\leftarrow$  d
      nodoAux.nivel++
       $res \leftarrow$  nodoAux
    else
       $res \leftarrow d$ 
    end if
  end if
end function

```

---

---

**Algorithm 36** Implementación de Borrar

---

```

function IBORRAR(inout d: estr, in c:  $\kappa$ )
  if  $d == NULO$  then
    endFunction
  else if  $c > d.clave$  then
     $d.derecho \leftarrow iBorrar(d.derecho, c)$ 
  else if  $c < d.clave$  then
     $d.izquierdo \leftarrow iBorrar(d.izquierdo, c)$ 
  else if  $d.izquierdo == NULO \wedge d.derecho == NULO$  then
     $d \leftarrow NULO$ 
  else if  $d.izquierdo == NULO$  then
     $aux \leftarrow d.derecho$ 
    while  $aux.izquierdo \neq NULO$  do
       $aux \leftarrow aux.izquierdo$ 
    end while
     $d.derecho \leftarrow iBorrar(aux.clave, d.derecho)$ 
     $d.clave \leftarrow aux.clave$ 
  else
     $aux \leftarrow d.izquierdo$ 
    while  $aux.derecho \neq NULO$  do
       $aux \leftarrow aux.derecho$ 
    end while
     $d.izquierdo \leftarrow iBorrar(aux.clave, d.izquierdo)$ 
     $d.clave \leftarrow aux.clave$ 
  end if
   $d \leftarrow Nivelar(T)$ 
   $d \leftarrow Torsion(T)$ 
  if  $d.derecho \neq NULO$  then
     $(d.derecho).derecho \leftarrow Torsion((d.derecho).derecho)$ 
  end if
   $d \leftarrow Division(T)$ 
   $d.derecho \leftarrow Division(d.derecho)$ 
   $res \leftarrow d$ 
end function
procedure NIVELAR(inout d: estr)
   $nivel\_correcto \leftarrow \min((d.izquierdo).nivel, (d.derecho).nivel) + 1$ 
  if  $nivel\_correcto < d.nivel$  then
     $d.nivel \leftarrow nivel\_correcto$ 
    if  $nivel\_correcto < (d.derecho).nivel$  then
       $(d.derecho).nivel \leftarrow nivel\_correcto$ 
    end if
  end if
end procedure

```

---