

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		

Contents

1	Módulo Red	3
2	Módulo DCNet	12
3	Módulo Cola de Prioridad $_{HEAP}$ (α)	20
4	Módulo Diccionario $_{TRIE}$ (α)	25
5	Módulo Diccionario $_{AVL}$ (κ, σ)	26

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

Descripción: conectar dos computadoras de la red.

VECINOS(**in** r : red, **in** c : hostname) $\rightarrow res$: conj(hostname)

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

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

Complejidad: O(n*L)

Descripción: da el conjunto de computadoras vecinas.

Aliasing: el conjunto se devuelve por copia.

USAINTERFAZ?(in r : red, in c : hostname, in i : interfaz) $\rightarrow res$: bool

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

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 : red, in c_1 : hostname, in c_2 : hostname) $\rightarrow res$: conj(lista(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 : red, in c_1 : hostname, in c_2 : hostname) $\rightarrow res$: 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 : red, in r_2 : red) $\rightarrow res$: 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 : red) $\rightarrow res$: 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 $\forall r$: red, $\forall cc$: conj(compu), $\forall s$: hostname, $\forall cs$: conj(secu(compu)), $\forall secu$: secu(compu)

```

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

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

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

dameSecuDeHostnames(secu)  $\equiv$  if vacia?(secu) then
     $\langle \rangle$ 
else
    ip(prim(secu))  $\bullet$  dameSecuDeHostnames(fin(secu))
fi

```

Fin TAD

Representación

red se representa con estr_red

donde:

estr_red es dicc(hostname, datos)

donde datos es tupla(*interfaces*: conj(interfaz)

conexiones: dicc(interfaz, hostname)

alcanzables: dicc(dest: hostname, caminos: conj(lista(hostname))))

hostname es string, interfaz es nat.

Rep : estr_red \longrightarrow bool

```

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

```

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

```

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

```

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

\\ Rep en Castellano:

Para cada computadora:

1: Las interfaces usadas pertenecen al conjunto de interfaces de la compu.

2: Los vecinos pertenecen 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.

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

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

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

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

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

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

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

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)
end function

```

Algorithm 7 Implementación de Vecinos

```

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

```

Algorithm 8 Implementación de UsaInterfaz

```

function iUSAIINTERFAZ(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 9 Implementación de CaminosMinimos

```

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

```

Algorithm 10 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 11 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 ¬(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
                                end if
                                Avanzar(itCaminos1)             ▷ O(1)
                            end while
                        end if
                    end if
                end while
            end if
        end if
    end function

```

Algorithm 12 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), DICT_{TRIE}(κ , σ), DICT_{AVL}(κ , σ), CONJ_{HEAP}(α), ITCONJ_{HEAP}(α).

se explica con: DCNET.

géneros: dcnet.

Operaciones de DCNet

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

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

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

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

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

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

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

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

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

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

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

LAQUEMASENVIO(in $d: \text{dcnet}$) $\rightarrow res: \text{hostname}$

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

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

Descripción: devuelve la computadora que más paquetes envió.

Aliasing: res se devuelve por copia.

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

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

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

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

TAD DCNET EXTENDIDA

extiende DCNET

otras operaciones

(no exportadas)

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
 Ag(ip(dameUno(cc)), dameHostnames(sinUno(cc)))
fi
 dameCompu(d, s) \equiv auxDameCompu($s, \text{computadoras}(\text{red}((d)))$)
 auxDameCompu(s, cc) \equiv **if** ip(dameUno(cc)) = s **then**
 dameUno(cc)
else
 auxDameCompu($s, \text{sinUno}(cc)$)
fi

```

dameSecuDeHostnames(secu)  $\equiv$  if vacia?(secu) then
    <>
else
    ip(prim(secu)) • dameSecuDeHostnames(fin(secu))
fi
IDpaqueteEnTransito?(d, p)  $\equiv$  auxIDpaqueteEnTransito(d, computadoras(red(d)), p)
auxIDpaqueteEnTransito(d, cc, p)  $\equiv$  if vacio?(cc) then
    false
else
    if p  $\in$  dameIDpaquetes(enEspera(dameUno(cc))) then
        true
    else
        auxIDpaqueteEnTransito(d, sinUno(cc), p)
    fi
fi
dameIDpaquetes(cp)  $\equiv$  if vacio?(cp) then
     $\emptyset$ 
else
    Ag( id(dameUno(cp)), dameIDpaquetes(sinUno(cp)) )
fi

```

Fin TAD

Representación

dcnet se representa con *estr_dcnet*

donde *estr_dcnet* es tupla(*red*: red
computadoras: dicc(hostname, X)
porHostname: dicc_{TRIE} (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: conj_{HEAP}(itConj(paquete))
paqPorID: dicc_{AVL} (IDpaquete, itConj(paquete))
cantEnvios: nat)

Rep : *estr_dcnet* \longrightarrow bool

Rep(*e*) \equiv true \iff ...

Abs : *estr_dcnet d* \longrightarrow dcnet {Rep(*d*)}

Abs(*d*) \equiv ...

Algoritmos

Algorithm 13 Implementación de Red

```

function iRED(in  $d$ : estr_dcnet)  $\rightarrow$  res: Red
  res  $\leftarrow$  d.red
end function

```

Algorithm 14 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

```

Algorithm 15 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

```

Algorithm 16 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 17 Implementación de iniciarDCNet

```

function IINICIARDCNET(in r: red) → res: estr_dcnet
    diccCompus ← Vacio()                                             ▷ O(1)
    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))
        X ← <index, Vacio(), Vacio(), Vacio(), 0>                    ▷ O(n) segun el apunte
        itX ← Definir(diccCompus, Siguiente(itHostname), X)        ▷ O(L) + copy(X)
        Definir(diccHostname, Siguiente(itHostname), itX)          ▷ O(copy(Siguiente(itHostname) + copy(X))
        index ← index + 1                                             ▷ O(1)
        Avanzar(itHostname)                                           ▷ O(1)
    end while
    n ← #Claves(diccCompus)                                           ▷ O(1)
    arrayCaminos ← CrearArreglo(n)                                   ▷ O(n)
    itPC ← CrearIt(diccCompus)                                         ▷ O(1)
    itPC2 ← CrearIt(diccCompus)                                         ▷ O(1)
    while HaySiguiente(itPC) do                                       ▷ O(#Computadoras(r))
        arrayDestinos ← CrearArreglo(n)                             ▷ O(n)
        while HaySiguiente(itPC2) do                                  ▷ O(#Computadoras(r))
            itConj ← CaminosMinimos(r, SiguienteClave(itPC), SiguienteClave(itPC2)) ▷ O(???)
            if HaySiguiente(itConj) then                               ▷ O(1)
                arrayDestinos[SiguienteSignificado(itPC2).indice] ← Siguiente(itConj) ▷ O(1)
            else
                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 18

```

function CREARPAQUETE(in/out d: estr_dcnet, in p: paquete)
    itPC ← Significado(d.porHostname, paquete.origen)                ▷ O(L)
    itPq ← Agregar(SiguienteSignificado(itPC).paquetes, p)          ▷ O(copy(p))
    Encolar(SiguienteSignificado(itPC).cola, p.prioridad, itPq)     ▷ O(log(n)), n cantidad de nodos
    Definir(SiguienteSignificado(itPC).paquetesPorID, IDpaquete, itPq) ▷ O(log(n)), n cantidad de
    nodos
end function

```

Algorithm 19 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 20 Implementación de LaQueMasEnvio

```

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

```

Algorithm 21 Implementación de AvanzarSegundo

```

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

  \\ Inicializo Iterador
  itCompu ← crearIt(d.computadoras)                                            ▷ O(1)
  i ← 0
                                ▷ Ciclo 1: Desencolo y guardo en arreglo auxiliar.
  while (HaySiguiente(itCompu)) do                                            ▷ el ciclo se hará a lo sumo n veces
    if (¬(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 ==

```

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, α .

se explica con: COLA DE PRIORIDAD ALTERNATIVA.

géneros: heap.

Operaciones de Cola de Prioridad $_{HEAP}$

VACIA $(\rightarrow res : \text{heap})$

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

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

Complejidad: $O(1)$

Descripción: Crea una cola vacía.

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

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

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

Complejidad: $O(1)$

Descripción: Indica si la cola esta vacía.

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

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

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

Complejidad: $O(1)$

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

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

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

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

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

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

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

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: $O(\log(n))$

Descripción: Devuelve al elemento de mayor prioridad y lo remueve de la cola. La cola no debe estar vacía.

TAD COLA DE PRIORIDAD ALTERNATIVA(α)

géneros colaPrio(α)

exporta colaPrio(α), generadores, observadores

usa BOOL, NAT, TUPLA

observadores básicos

vacía? : colaPrior(α) \longrightarrow bool

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

desencolar : colaPrior(α) $c \longrightarrow$ colaPrior(α) $\{ \neg \text{vacía?}(c) \}$

generadores

$\text{vacía} : \longrightarrow \text{colaPrior}(\alpha)$
 $\text{encolar} : \text{nat} \times \alpha \times \text{colaPrior}(\alpha) \longrightarrow \text{colaPrior}(\alpha)$
axiomas $\forall c: \text{colaPrior}(\alpha), \forall e: \alpha$
 $\text{vacía?}(\text{vacía}) \equiv \text{true}$
 $\text{vacía?}(\text{encolar}(p, e, c)) \equiv \text{false}$
 $\text{próximo}(\text{encolar}(p, e, c)) \equiv \mathbf{if} \text{ vacía?}(c) \vee_{\text{L}} \Pi_1(\text{próximo}(c)) < p \mathbf{ then } < p, e > \mathbf{ else if } \Pi_1(\text{próximo}(c)) = p \mathbf{ then } < p, e > \vee \text{próximo}(c) \mathbf{ else } \text{próximo}(c) \mathbf{ fi}$
 $\text{desencolar}(\text{encolar}(p, e, c)) \equiv \mathbf{if} \text{ vacía?}(c) \vee_{\text{L}} \Pi_1(\text{próximo}(c)) < p \mathbf{ then } c \mathbf{ else if } \Pi_1(\text{próximo}(c)) = p \mathbf{ then } c \vee \text{encolar}(p, e, \text{desencolar}(c)) \mathbf{ else } \text{encolar}(p, e, \text{desencolar}(c)) \mathbf{ fi fi}$

Fin TAD

Representación

heap se representa con *estr*

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

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

Especificación de las operaciones auxiliares utilizadas para Rep y Abs

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

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

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

$\text{caminoHastaRaiz}(n) \equiv \text{if } n.\text{padre} = \text{null} \text{ then } 0 \text{ else } \text{caminoHastaRaiz}(n.\text{padre}) + 1$

Algoritmos

Algorithm 23 Implementación de Vacía

function $\text{IVACIA} \rightarrow \text{res: heap}$

$\text{res} \leftarrow \langle 0, \text{null} \rangle$

$\triangleright O(1)$

end function

Algorithm 24 Implementación de Vacía?

function $\text{IVACIA?}(\text{in } \text{estr: heap}) \rightarrow \text{res: bool}$

$\text{res} \leftarrow (\text{estr.primero} == \text{null})$

$\triangleright O(1)$

end function

Algorithm 25 Implementación de Primero

```

function iPRIMERO(in estr: heap)  $\rightarrow$  res: tupla(nat,  $\alpha$ )
    res  $\leftarrow$   $\langle$  (estr.primero).prioridad, (estr.primero).valor  $\rangle$   $\triangleright$  O(1)
end function

```

Algorithm 26 Implementación de Encolar

```

function iENCOLAR(in/out estr: heap, in prio: nat, in valor:  $\alpha$ )  $\rightarrow$  res: bool
    res  $\leftarrow$  true  $\triangleright$  O(1)
    if estr.size == 0 then  $\triangleright$  O(1)
        estr.primero  $\leftarrow$   $\langle$  null, null, null, prio, valor  $\rangle$   $\triangleright$  O(1)
    else
        size ++  $\triangleright$  O(1)
        x  $\leftarrow$  valor  $\triangleright$  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 árbol 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.primero  $\triangleright$  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(1)
        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.primero  $\wedge_L$  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(1)
            w.valor  $\leftarrow$  (w.padre).valor  $\triangleright$  O(1)
            (w.padre).valor  $\leftarrow$  aux  $\triangleright$  O(1)
            w  $\leftarrow$  w.padre  $\triangleright$  O(1)
        end while
    end if
end function

```

Algorithm 27 Implementación de Desencolar

```

function IDESENCOLAR(in/out estr: heap)  $\rightarrow$  res:  $\alpha$ 
    res  $\leftarrow$  estr.primer                                 $\triangleright O(1)$ 
    x  $\leftarrow$  valor                                        $\triangleright O(1)$ 
    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  $\rangle$               $\triangleright O(1)$ 
    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
    (estr.primer).valor  $\leftarrow$  z.valor                      $\triangleright O(1)$ 
    borrar(z)                                               $\triangleright O(1)$ 
    size  $\leftarrow$  —                                          $\triangleright O(1)$ 
    while (z.izq  $\neq$  null  $\vee$  z.der  $\neq$  null)  $\wedge_L$  z.valor < maxValor(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  $O(\log(n))$ 
         $\triangleright$  maxValor devuelve el maximo valor si ambos punteros son validos, o el valor apuntado por el
        puntero no nulo en caso de que alguno no lo sea
        if z.der == null  $\vee_L$  (z.izq).valor  $\geq$  (z.der).valor then  $\triangleright O(1)$ 
            aux  $\leftarrow$  z.valor                                 $\triangleright O(1)$ 
            z.valor  $\leftarrow$  (z.izq).valor                     $\triangleright O(1)$ 
            (z.izq).valor  $\leftarrow$  aux                         $\triangleright O(1)$ 
            z  $\leftarrow$  z.izq                                   $\triangleright O(1)$ 
        else
            aux  $\leftarrow$  z.valor                                 $\triangleright O(1)$ 
            z.valor  $\leftarrow$  (z.der).valor                     $\triangleright O(1)$ 
            (z.der).valor  $\leftarrow$  aux                         $\triangleright O(1)$ 
            z  $\leftarrow$  z.der                                   $\triangleright O(1)$ 
        end if
    end while
end function

```

4 Módulo Diccionario $TRIE$ (α)

Interfaz

usa: .

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO).

Operaciones

DEFINIDA(**in** d : dicc $_{TRIE}$ (clave, significado), **in** c : clave) $\rightarrow res$: Bool

Pre $\equiv \{true\}$

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

Complejidad: $O(L)$

OBTENER(**in** d : dicc $_{TRIE}$ (clave, significado), **in** c : clave) $\rightarrow res$: significado

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

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

Complejidad: $O(L)$

VACIO() $\rightarrow res$: dicc $_{TRIE}$ (clave, significado)

Pre $\equiv \{true\}$

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

Complejidad: $O(1)$

DEFINIR(**in/out** d : dicc $_{TRIE}$ (clave, significado), **in** c : clave, **in** s : significado) $\rightarrow res$: Bool

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

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

Complejidad: $O(L)$

BORRAR(**in/out** d : dicc $_{TRIE}$ (clave, significado), **in** c : clave) $\rightarrow res$: Bool

Pre $\equiv \{d=d_0\}$

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

Complejidad: $O(L)$

Descripción: Devuelve TRUE si se pudo borrar la clave, o FALSE si no la encontró

CLAVES(**in** d : dicc $_{TRIE}$ (clave, significado)) $\rightarrow res$: conj(clave)

Pre $\equiv \{true\}$

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

Complejidad: $O()$

5 Módulo Diccionario $_{AVL} (\kappa, \sigma)$

Interfaz

usa: .

se explica con: DICCIONARIO(CLAVE,SIGNIFICADO).

géneros: $\text{dicc}_{AA}(\text{clave}, \text{significado})$.

Operaciones

DEFINIDO?(**in** $d: \text{dicc}_{AA}(\text{clave}, \text{significado})$, **in** $c: \text{clave}$) $\rightarrow res: \text{Bool}$

Pre $\equiv \{\}$

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

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

SIGNIFICADO(**in** $d: \text{dicc}_{AA}(\text{clave}, \text{significado})$, **in** $c: \text{clave}$) $\rightarrow res: \text{significado}$

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

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

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

VACIO() $\rightarrow res: \text{dicc}_{AA}(\text{clave}, \text{significado})$

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

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

Complejidad: $O(1)$

DEFINIR(**in/out** $d: \text{dicc}_{AA}(\text{clave}, \text{significado})$, **in** $c: \text{clave}$, **in** $s: \text{significado}$)

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

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

Complejidad: $O()$

BORRAR(**in/out** $d: \text{dicc}_{AA}(\text{clave}, \text{significado})$, **in** $c: \text{clave}$)

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

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

Complejidad: $O()$

Representación

dicc_{AA} se representa con estr_{AA}

donde estr_{AA} es $\text{tupla}(id: \text{nat}$
 $\text{izquierdo: puntero}(\text{nodo}(id, it))$
 $\text{derecho: puntero}(\text{nodo}(id, it))$
 $\text{valor: itConj}(\alpha)$
 $)$

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

$\text{Rep}(e) \equiv \text{true} \iff \dots$

$\text{Abs} \quad \quad \quad : \text{estr } r \rightarrow \text{dicc}_{AA}(id, it) \quad \quad \quad \{\text{Rep}(r)\}$

$\text{Abs}(r) \equiv \dots$

Algoritmos

Algorithm 28 Implementación de Definido?

```
function iDEFINIDO?(in d: estr_AA , in c: clave)→ res: bool
  nodoActual ← d
  res ← FALSE
  while ¬(nodoActual == NULO) && ¬res do
    if nodoActual.id == c then res ← TRUE
    else
      if c < nodoActual.id then nodoActual ← nodoActual.izquierdo
      else nodoActual ← nodoActual.derecho
    end if
  end if
end while
end function
```

Algorithm 29 Implementación de Significado

```
function iSIGNIFICADO(in d: estr_AA , in c: clave)→ res:  $\alpha$ 
  nodoActual ← d
  res ← FALSE
  while ¬(nodoActual == NULO) && ¬res do
    if nodoActual.id == c then res ← nodoActual.valor
    else
      if c < nodoActual.id then nodoActual ← nodoActual.izquierdo
      else nodoActual ← nodoActual.derecho
    end if
  end if
end while
end function
```

Algorithm 30 Implementación de Vacio

```
function iVACIO→ res: estr_AA
  res ← tupla(0, NULO, NULO, NULO, 0)
end function
```

Algorithm 31 Implementación de Definir

```

function IDEFINIR(in d: estr_AA , in c: nat, in s:  $\alpha$ )
  nodoActual  $\leftarrow$  d
  yaDefini  $\leftarrow$  FALSE
  while  $\neg$ yaDefini do
    if  $c < \text{nodoActual.id}$  then
      if  $\text{nodoActual.izquierdo} == \text{NULO}$  then
        nuevoNodo  $\leftarrow$  Vacio()
        nuevoNodo.valor  $\leftarrow$  Copiar(s)
        nuevoNodo.id  $\leftarrow$  c
        nodoActual.izquierdo  $\leftarrow$  nuevoNodo
        yaDefini gets TRUE
      else
        nodoActual  $\leftarrow$  nodoActual.izquierdo
      end if
    else
      if  $\text{nodoActual.derecho} == \text{NULO}$  then
        nuevoNodo  $\leftarrow$  Vacio()
        nuevoNodo.valor  $\leftarrow$  Copiar(s)
        nuevoNodo.id  $\leftarrow$  c
        nodoActual.derecho  $\leftarrow$  nuevoNodo
        yaDefini  $\leftarrow$  TRUE
      else
        nodoActual  $\leftarrow$  nodoActual.derecho
      end if
    end if
  end while
  nodoActual  $\leftarrow$  Torsion(nodoActual)
  nodoActual  $\leftarrow$  Division(nodoActual)
end function

```

Algorithm 32 Implementación de Torsion

```

function ITORSION(in d: estr_AA)  $\rightarrow$  res: estr_AA
  if  $d == \text{NULO} \parallel d.\text{izquierdo} == \text{NULO}$  then
    res  $\leftarrow$  d
  else
    if  $\text{Altura}(d.\text{izquierdo}) \geq \text{Altura}(d)$  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 33 Implementación de Division

```
function IDIVISION(in d: estr_AA)→ res: estr_AA
  if d == NULO || d.derecho == NULO || d.derecho.derecho == NULO then
    res ← d
  else
    if Altura(d.derecho.derecho) ≥ Altura(d) then
      nodoAux ← d.derecho
      d.derecho ← nodoAux.izquierdo
      nodoAux.izquierdo ← d
      nodoAux.altura ← nodoAux.altura + 1
      res ← nodoAux
    else
      res ← d
    end if
  end if
end function
```

Algorithm 34 Implementación de Altura

```
function IALTURA(in d: estr_AA)→ res: nat_0
  if d.derecho == NULO && d.izquierdo == NULO then
    res ← 1
  else
    if d.derecho == NULO then
      res ← Altura(d.izquierdo) + 1
    else
      if d.izquierdo == NULO then
        res ← Altura(d.derecho) + 1
      else
        Minimo(Altura(d.izquierdo), Altura(d.derecho))
        ▷ el mínimo de dos nat
      end if
    end if
  end if
end function
```
