DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2: Diseño

Primer cuatrimestre - 2015 Algoritmos y Estructuras de Datos II

Grupo 2

Integrante	LU	Correo electrónico
Benitez, Nelson	945/13	nelson.benitez92@gmail.com
Roizman, Violeta	273/11	violeroizman@gmail.com
Vázquez, Jésica	318/13	jesis_93@hotmail.com
Zavalla, Agustín	670/13	nkm747@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria – Pabellón I (Planta Baja)
Intendente Güiraldes 2160 – C1428EGA
Ciudad Autónoma de Buenos Aires – Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1 DCNet

Una DCNet es un sistema que tiene computadoras en red que reciben paquetes que envían a la computadora destino a cada segundo.

1.1 Interfaz

se explica con DCNet

usa Compu, Paquete, Red, diccPref, conjLog, conjLogP

géneros `dcnet`

Operaciones

crearSistema(**in** r : `red`) $\longrightarrow$ res : `dcnet`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} iniciarDCNet(r)\}$
Descripción: Crea un sistema DCNet.
Complejidad: O($\max\{n^2 \times complejidad(caminominimo), n \times L\}$)
Aliasing: el campo.red del modulo devuelto es un puntero a la red de entrada

crearPaquete(**in/out** s : `dcnet`, **in** p : `paquete`)
Pre $\equiv$ $\{s =_{\text{obs}} s_0 \wedge (\forall p_0 : paquete, paqueteEnTransito?(p, s))\ \neg(p_0 =_{\text{obs}} p) \wedge$
$destino(p) \in compus(red) \wedge origen(p) \in compus(red) \wedge_{\text{L}}$
$haycamino?(destino(p), origen(p), red(s))\}$
Post $\equiv$ $\{s =_{\text{obs}} crearPaquete(s_0, p)\}$
Descripción: Crea un paquete y lo agrega a la computadora correspondiente.
Complejidad: O($L + log(k)$)

avanzarSegundo(**in/out** s : `dcnet`)
Pre $\equiv$ $\{s =_{\text{obs}} s_0\}$
Post $\equiv$ $\{s =_{\text{obs}} avanzarSegundo(s_0)\}$
Descripción: Avanza un segundo el sistema. Todas las computadoras envían su respectivo paquete y en consecuencia se actualizar los paquetes en espera de cada una de ellas.
Complejidad: O($n \times\ (L + log(k))$)
Aliasing:

dameRed(**in** s : `dcnet`) $\longrightarrow$ res : `puntero(red)`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} red(s)\}$
Descripción: Devuelve la red de DCNet.
Complejidad: O(1)
Aliasing: Devuelve un puntero a la misma red que la que se pasó como parámetro para crear el sistema

caminoRecorrido(**in** s : `dcnet`, **in** p : `paquete`) $\longrightarrow$ res : `secu(compu)`
Pre $\equiv$ $\{paqueteEnTransito?(s, p)\}$
Post $\equiv$ $\{res =_{\text{obs}} caminoRecorrido(s, p)\}$
Descripción: Devuelve el camino recorrido hasta el momento por un paquete.
Complejidad: O($n \times\ log(max(n, k))$)
Aliasing: Devuelve puntero al camino recorrido que se encuentra en CaminosMinimos

CANTIDADENVIADOS(**in** s : `dcnet`, **in** c : `compu`) $\longrightarrow$ res : `nat`
Pre $\equiv \{c \in computadoras(red(s))\}$
Post $\equiv \{res =_{\text{obs}} cantidadEnviados(s, c)\}$
Descripción: Devuelve la cantidad de paquetes enviados por una computadora.
Complejidad: O(n)

ENESPERA(**in** s : `dcnet`, **in** c : `compu`) $\longrightarrow$ res : `puntero(conjLogP(paquete))`
Pre $\equiv \{c \in computadoras(red(s))\}$
Post $\equiv \{res =_{\text{obs}} enEspera(s, c)\}$
Descripción: Devuelve un iterador a los paquetes de la computadora.
Complejidad: O(L)
Aliasing: Hay aliasing entre res y el conjunto de paquetes de la computadora pasada por parámetro

LAQUEMASENVIO(**in** s : `dcnet`) $\longrightarrow$ res : `compu`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} laQueMasEnvio(s, p)\}$
Descripción: Devuelve la computadora que más paquetes envió
Complejidad: O(1)
Aliasing:

PAQUETEENTRANSITO?(**in** s : `dcnet`, *in* p : `paquete`) $\longrightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv \{res = paqueteEnTransito?(s, p)\}$
Descripción: Devuelve true si el paquete se encuentra en transito en el sistema
Complejidad: O($n \times log(k)$)

Las complejidades están en función de las siguientes variables:
n : la cantidad total de computadoras que hay en el sistema,
L : el hostname más largo de todas las computadoras,
k : la cola de paquetes más larga de todas las computadoras.

1.2 Representación

se representa con `sistema`

donde `sistema` **es** `tupla`⟨Compus : `arreglo(tupla`⟨IP : `String`,),
pN : `puntero(conjLog(paquete))`,
pN' : `puntero(conjLog(paquetePos))`,
#Paquetes : `nat`⟩
CompusPorPref : `diccPref(compu,tupla`⟨PorNom : `conjLog(paquete)`, ,
PorPrior : `conjLog(paquete)`,
PorNom' : `conjLog(paquetePos)`,
PorPrior' : `conjLog(paquetePos)`⟩
CaminosMinimos : `arreglo(arreglo(arreglo(compu)))`,
LaQMasEnvio : `nat` ,
Red : `red`⟩

esto se puede borrar despues: aclaracion en compus en cada indice del arreglo esta la compu correspondiente a esa numeracion

Invariante de representación

1. Todos los IP de *compus* pertenecen al conjunto de claves de *CompusPorPref* y la longitud de dicho arreglo es igual al cardinal de las claves del diccionario.
2. Los pN de las tuplas que tiene el arreglo *compus* apuntan al conjunto de paquetes(PorNom) de un significado en *CompusPorPref* cuya clave es igual al IP de esa posición en el arreglo.
3. Todos los conjuntos de los significados de *CompusPorPref* son disjuntos dos a dos.
4. Los conjuntos de los campos de la tupla PorNom, PorPrior son iguales.
5. La longitud de *CaminosMinimos* es igual a la longitud del arreglo que tiene *CaminosMinimos* en cada posición.
6. La longitud del arreglo, que tiene un arreglo de *CaminosMinimos* es menor o igual a la longitud de *CaminosMinimos*.
7. Los elementos del arreglo anteriormente mencionado son IPs del diccionario *CompusPorPref* y no tiene repetidos.
8. La computadora que más paquetes envió es aquella cuyo índice es igual a *LaQMasEnvio*

Rep : $\widehat{\texttt{sistema}} \longrightarrow$ *boolean*

$(\forall s : \widehat{\texttt{sistema}})$
Rep$(s) \equiv$
1. $\forall s$: String $def?(s, s.CompusPorPref), (\exists c : compu), esta?(c, s.Compus) \ \wedge \ \pi_1(c) = s \ \wedge$ longitud$(s.Compus) =$ #CLAVES$(s.CompusPorPref)$
2. $\forall c$: compu esta?$(c, s.Compus), *\pi_2(c) =$ obtener$(\pi_1(c), s.CompusPorPref)$
3. $\forall s, t$: String def?$(s, s.CompusPorPref) \ \wedge$ def?$(t, s.CompusPorPref) \ \wedge \ s \neq t \ \Rightarrow_{\text{L}}$ obtener$(s, s.CompusPorPref) \bigcap$ obtener$(t, s.CompusPorPref) = \emptyset$
4. $\forall s$: String def?$(s, s.CompusPorPref) \ \Rightarrow_{\text{L}} \ \pi_1($obtener$(s, s.CompusPorPref)) =$ $\pi_2($obtener$(s, s.CompusPorPref))$
5, 6, 7. $(\forall i, j$: nat$), \ 0 \leq i, j <$ longitud$(s.CaminosMinimos) \ \Rightarrow_{\text{L}}$longitud$(s.CaminosMinimos) =$ longitud$(s.CaminosMinimos[i]) \wedge$ longitud$(s.CaminosMinimos[i][j]) <$ longitud$(s.CaminosMinimos) \wedge$ $(\forall e$: nat$)$, esta?$(e, s.CaminosMinimos[i][j]) \Rightarrow$ pertenece$(e, s.CompusPorPref)$
8.$\forall c$: compu esta?$(c, s.Compus) \ \Rightarrow_{\text{L}} \ \pi_3(c) \leq \pi_3(s.Compus[s.LaQMasEnvio])$

Función de abstracción

Abs : $\widehat{\texttt{dcnet}} \ s \longrightarrow \widehat{\texttt{DCNet}}$ $\{$Rep$(s)\}$

$(\forall s : \widehat{\texttt{dcnet}})$
Abs$(s) \equiv dc : \widehat{\texttt{DCNet}} \mid$
$red(dc) =^*(s.red) \wedge (\forall c : compu, c \in compus(dc))(enEspera(dc, c) =^*(enEspera(s, c)) \wedge$
$cantidadEnviados(dc, c) = cantidadEnviados(s, c)) \wedge$
$(\forall p : paquete, paqueteEnTransito?(dc, p))caminoRecorrido(dc, p) =^*(caminoRecorrido(s, p))$

1.3 Algoritmos

ICREARSISTEMA(**in** r : `red`) $\longrightarrow$ res : `dcnet`	
$res.red \leftarrow r$	
$n \leftarrow \#(\text{COMPUS}(red))$	O(#compus(red)=n)?
$i \leftarrow 0$	
$j \leftarrow 0$	O(1)
$res.Compus \leftarrow \text{CREARARREGLO}(n)$	O(n)
$res.CaminosMinimos \leftarrow \text{CREARARREGLO}(n)$	O(n)
var p : `arreglo_dimensionable de puntero(conjLog(paquete))`	
while i<n **do**	O(n)
$res.CaminosMinimos[i] \leftarrow \text{CREARARREGLO}(n)$	O(n)
s: $< nat, conjLog(paquete, <_{id}), conjLog(paquete, <_p),$ $conjLog(paquetePos, <_{id}), conjLog(paquetePos, <_p) >$	
$\pi_1(s) \leftarrow compu(r, i)$	
$\pi_2(s) \leftarrow \text{NUEVO}()$	
$\pi_3(s) \leftarrow \text{NUEVO}()$	
$\pi_4(s) \leftarrow \text{NUEVO}()$	
$\pi_5(s) \leftarrow \text{NUEVO}()$	
DEFINIR(res.CompusPorPref,compu(r,i),s)	O(L)
$p[i] \leftarrow pi_3(s)$	
$p'[i] \leftarrow pi_5(s)$	
$res.Compus[i] \leftarrow < compu(r, i), p[i], p'[i], 0 >$	O(1)
while j<n **do**	O(n)
$res.CaminosMinimos[i][j] \leftarrow caminoMinimo(compu(r, i), compu(r, j), r)$	O(complejidad cammin(red))
$j++$	
end while	
$i++$	
end while	
$res.LaQMasEnvio \leftarrow 0$	O(1)
	O($\max\{n^5, n \times L\}$)

ICREARPAQUETE(**in/out** s : `dcnet`, **in/out** p : `paquete`)	
$t_1 : < nat, conjLog(paquete, <_{id}), conjLog(paquete, <_p),$ $conjLog(paquetePos, <_{id}), conjLog(paquetePos, <_p) >$	
$t_1 \leftarrow \text{OBTENER}(origen(p),\ s.CompusPorPref)$	O(L)
$t_2 : < nat, conjLog(paquete, <_{id}), conjLog(paquete, <_p),$ $conjLog(paquetePos, <_{id}), conjLog(paquetePos, <_p) >$	
$t_2 \leftarrow \text{OBTENER}(destino(p),\ s.CompusPorPref)$	O(L)
p': $paquetePos$	
$indiceOrigen(p') \leftarrow \pi_1(t_1)$	O(1)
$indiceDestino(p') \leftarrow \pi_1(t_2)$	O(1)
$indiceActual(p') \leftarrow 0$	
$\text{INSERTAR}(\pi_2(t), p)$	O($log(k)$)
$\text{INSERTAR}(\pi_3(t), p)$	O($log(k)$)
$\text{INSERTAR}(\pi_4(t), p')$	O($log(k)$)
$\text{INSERTAR}(\pi_5(t), p')$	O($log(k)$)
	O($L + log(k)$)

ILAQUEMASENVIO(**in** s : `dcnet`) $\longrightarrow$ res : `compu`	
$res \leftarrow s.compus[s.LaQMasEnvio].IP$	O(1)

	O(1)

IDAMERED(**in** s : `dcnet`) $\longrightarrow$ res : `puntero(red)`

$res \leftarrow \&(s.red)$	O(1)
	O(1)

IENESPERA(**in** s : `dcnet`, **in** c : `compu`) $\longrightarrow$ res : `puntero(conjLogP(paquete))`

$t : < nat, conjLog(paquete, <_{id}), conjLog(paquete, <_p),$ $conjLog(paquetePos, <_{id}), conjLog(paquetePos, <_p) >$	
$t \leftarrow$ OBTENER($\pi_1(c)$, $s.CompusPorPref$)	O(L)
$res \leftarrow \&(\pi_3(t))$	O(1)
	O(L)

IAVANZARSEGUNDO(**in/out** s : `dcnet`)

var i : `nat`	
$i \leftarrow 0$	O(1)
var m : `nat`	
$m \leftarrow s.Compus[LaQMasEnvio].\#PaqE$	
while $i <$ LONGITUD($s.Compus$) **do**	O(n)
var $paqYProxDes$: `arreglo_dimensionable de tupla de paquetePos y nat`	
$paqYProxDes \leftarrow$ CREARARREGLO(n)	
var IP : `String`	
$IP \leftarrow s.Compus[i].IP$	
$t_1 : < nat, conjLog(paquete, <_{id}), conjLog(paquete, <_p),$ $conjLog(paquetePos, <_{id}), conjLog(paquetePos, <_p) >$	
$t_1 \leftarrow$ OBTENER($IP, s.CompusPorPref$)	O(L)
var p : `paquete`	
if $\neg$VACIA?($\pi_5(t_1)$) **then**	
$p' \leftarrow$ SACARMAX($\pi_5(t_1)$)	O($log(k)$)
BORRAR($\pi_2(t_1)$, PAQUETE(p'))	O($log(k)$)
BORRAR($\pi_3(t_1)$, PAQUETE(p'))	O($log(k)$)
BORRAR($\pi_4(t_1)$, p')	O($log(k)$)
BORRAR($\pi_5(t_1)$, p')	O($log(k)$)
$s.Compus[i].\#PaqE \leftarrow s.compus[i].\#PaqE + 1$	O(1)
$proxima \leftarrow s.CaminosMinimos$[INDICEORIGEN($p'$)][INDICEDESTINO($p'$)][INDICEACTUAL($p'$)+ 1]	O(L) (se copia)
if $s.Compus[i].\#PaqE > max$ **then**	O(1)
$max \leftarrow i$	O(1)
end if	
$paqYProxDes[i] \leftarrow < p', proxima >$	
else	
$paqYProxDes[i] \leftarrow 0$	
end if	
$i \leftarrow i + 1$	O(1)
end while	
$s.LaQMasEnvio \leftarrow max$	O(1)
$i \leftarrow 0$	O(1)
while $i <$ LONGITUD($s.Compus$) **do**	O(n)
if $paqYProxDes[i] \neq 0$ **then**	O(1)
$p' \leftarrow \pi_1(paqYProxDes[i])$	O(1)
$proxima \leftarrow \pi_2(paqYProxDes[i])$	O(L)
if $\neg$(DESTINO(PAQUETE(p')) $= proxima$) **then**	O(L)

ACTUALIZARINDICE(p')	O(1)
$t_2 :< nat, conjLog(paquete, <_{id}), conjLog(paquete, <_p),$ $conjLog(paquetePos, <_{id}), conjLog(paquetePos, <_p) >$	
$t_2 \leftarrow$ OBTENER($proxima, s.CompusPorPref$)	O(L)
INSERTAR($\pi_2(t_2)$, PAQUETE(p'))	O($log(k)$)
INSERTAR($\pi_3(t_2)$, PAQUETE(p'))	O($log(k)$)
INSERTAR($\pi_4(t_2)$, p')	O($log(k)$)
INSERTAR($\pi_5(t_2)$, p')	O($log(k)$)
end if	
end if	
end while	
	O($n \times (L + log(k))$)

ICANTIDADENVIADOS(**in/out** s : `dcnet`, **in** c : `compu`) $\longrightarrow$ res : `nat`

var i : `nat`	
$i \leftarrow 0$	O(1)
while $s.compus[i].IP \neq \pi_1(c)$ **do**	O(n)
$i \leftarrow i + 1$	O(1)
end while	
$res \leftarrow s.compus[i].\#PaqE$	O(1)
	O(n)

IPAQUETEENTRANSITO?(**in** s : `dcnet`, *in* p : `paquete`) $\longrightarrow$ res : `bool`

var i : `nat`	
$i \leftarrow 0$	O(1)
var b : `bool`	
$b \leftarrow \neg$(PERTENECE?(p,*($s.compus[i].pN$)))	O($log(k)$)
while $b \wedge i < n$ **do**	O(n)
$i \leftarrow i + 1$	O(1)
$b \leftarrow \neg$(PERTENECE?(p,*($s.compus[i].pN$)))	O($log(k)$)
end while	
if $i = n \wedge b$ **then**	O(1)
$res \leftarrow false$	O(1)
else	
$res \leftarrow true$	O(1)
end if	
	O($n \times log(k)$)

ICAMINORECORRIDO(**in** s : `dcnet`, **in** p : `paquete`) $\longrightarrow$ res : `secu(compu)`

var i : `nat`	
$i \leftarrow 0$	O(1)
var b : `bool`	
$b \leftarrow \neg$(PERTENECE?(p,*($s.compus[i].pN$)))	O($log(k)$)
while b **do**	O(n)
$i \leftarrow i + 1$	O(1)
$b \leftarrow \neg$(PERTENECE?(p,*($s.compus[i].pN'$)))	O($log(k)$)
end while	
$res \leftarrow s.CaminosMinimos$[INDICEORIGEN($p'$)][$i$]	O(n) (por copia)
	O($n \times log(k)$)

1.4 Servicios Usados

Del modulo ConjLog requerimos pertenece, buscar, sacarMax, insertar y borrar en O($log(k)$) .

Del modulo Diccionario Por Prefijos requerimos Def?, obtener en O(L).

2 Red

El módulo red permite crear una red de computadoras, agregar nuevas, conectarlas y averiguar el camino mínimo entre dos de ellas.

2.1 Interfaz

se explica con RED

géneros `red`

Operaciones

NUEVA() $\longrightarrow$ *res* : `red`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $\equiv$ iniciarRed}
Descripción: Crea una red vacía
Complejidad: O(1)
Aliasing:

INTERFAZUSADA(**in** r : `red`, *in* c : `compu`, *in c1* : `compu`) $\longrightarrow$ *res* : `interfaz`
Pre $\equiv$ $\{\neg(c = c1) \land c \in compus(r) \land c1 \in compus(r)\}$
Post $\equiv$ $\{res =_{\text{obs}} InterfazUsada(r, c, c1)\}$
Descripción: Retorna por copia la interfaz de la primer compu recibida por parámetro usada para conectarse con la segunda
Complejidad: O(#(compus(r)))
Aliasing:

COMPUS(**in** r : `red`) $\longrightarrow$ *res* : `conj(compu)`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} computadoras(r)\}$
Descripción: Devuelve el conjunto de compus
Complejidad: O(1)
Aliasing: Retorna el conjunto de computadoras por referencia

CONECTADAS?(**in** r : `red`, *in* c : `compu`, *in c1* : `compu`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ $\{\neg(c = c1) \land c \in compus(r) \land c1 \in compus(r)\}$
Post $\equiv$ $\{res =_{\text{obs}} conectadas?(r)\}$
Descripción: Indica si dos compus estan conectadas
Complejidad: O(#compus(r))

AGREGARCOMPU(**in/out** r : `red`, **in** c : `compu`)
Pre $\equiv$ $\{r \equiv r_0 \land (\forall\ c_1$:compu)$(c_1 \in$ computadoras$(r)) \Rightarrow$ IP$(c) \neq$ IP$(c_1)\}$
Post $\equiv$ $\{r \equiv$ agregarComputadora$(r, c)\}$
Descripción: Agrega a la red r la computadora c
Complejidad: O(n + copy(c))
Aliasing:

CONECTAR(**in/out** r : `red`, **in** *c1* : `compu`, **in** *i1* : `interfaz`, **in** *c2* : `compu`, **in** *i2* : `interfaz`)
Pre $\equiv$ $\{r \equiv r_0 \land c_1 \in$ computadoras$(r) \land c_2 \in$ computadoras$(r) \land$ IP(c1) $\neq$ IP(c2)
$\land \neg$conectadas(r,c1,c2) $\land \neg$UsaInterfaz?(r,c1,i1) $\land \neg$UsaInterfaz?(r,c2,i2)$\}$
Post $\equiv$ $\{r \equiv$ conectar$(r, c1, i1, c2, i2)\}$
Descripción: Conecta las computadoras c_1 y c_2

Complejidad: O(n)
Aliasing:

CaminosMinimos(**in** r : `red`, **in** *c1* : `compu`, **in** *c2* : `compu`) $\longrightarrow$ *res* : `conj(secu(nat))`
Pre $\equiv \{c_1 \in \text{computadoras}(r) \wedge c_2 \in \text{computadoras}(r)\}$
Post $\equiv \{r \equiv \text{caminosMinimos}(r,c_1,c_2)\}$
Descripción: Devuelve los caminos mínimos entre dos computadoras
Complejidad: O(?)
Aliasing:

CaminoMinimo(**in** r : `red`, *in* f : `compu`, *in* d : `compu`) $\longrightarrow$ *res* : `secu(compu)`
Pre $\equiv \{c_1 \in \text{computadoras}(r) \wedge c_2 \in \text{computadoras}(r)\}$
Post $\equiv \{r \equiv \text{caminosMinimos}(r,c_1,c_2)\}$
Descripción: Devuelve los caminos mínimos entre dos computadoras
Complejidad: O(n^5)
Aliasing:

StringAIndice(**in** r : `red`, **in** c : `compu`) $\longrightarrow$ *res* : `nat`
Pre $\equiv \{c \in \text{computadoras}(r)\}$
Post $\equiv \{res \equiv \text{indice}(r,c)\}$
Descripción: Devuelve un nat distinto para cada IP
Complejidad: O(n)
Aliasing:

UsaInterfaz?(**in** r : `red`, **in** c : `compu`, **in** i : `interfaz`) $\longrightarrow$ *res* : `bool`
Pre $\equiv \{c \in \text{computadoras}(r) \wedge i \in \text{interfaces}(c)\}$
Post $\equiv \{res \equiv UsaInterfaz?(r, c, i)\}$
Descripción: Devuelve TRUE si y solo si la interfaz está siendo utilizada
Complejidad: O(n)
Aliasing:

HayCamino?(**in** r : `red`, **in** *c1* : `compu`, **in** *c2* : `compu`) $\longrightarrow$ *res* : `bool`
Pre $\equiv \{c_1 \in \text{computadoras}(r) \wedge c_2 \in \text{computadoras}(r)\}$
Post $\equiv \{res \equiv HayCamino?(r, c1, c2)\}$
Descripción: Devuelve TRUE sí y solo sí hay un camino válido entre las dos computadoras
Complejidad: O(n^5)
Aliasing:

2.2 Representación

se representa con `redstr`
donde `redstr` **es** `tupla`⟨Compus : `conj(compu)`,
Conexiones : `dicc(compu,dicc(interfaz,compu))`⟩
donde `compu` **es** `tupla`⟨IP : `string`,
interfaces : `conj(interfaz)`⟩
donde `interfaz` **es** `nat`

2.3 Invariante de representación

1. El conjunto de claves de conexiones está incluído o es igual al conjunto de compus
2. 'Simetría en las conexiones' : Cada una de las claves de conexiones está en los significados de las compus a las que está conectada

3. Las compus conectadas a c están en las claves de las conexiones
4. Las claves del significado de cada una de las compus 'c' está incluída en el conjunto de interfaces de 'c'

Rep.

$claves(conexiones(r)) \subseteq compus(r) \wedge$

$((\forall c : compu)\ c \in claves(conexiones(r))) \Rightarrow_{\text{L}}$
$((\forall c_1 : compu) c_1 \in conectadas(conexiones(r), c)) \Rightarrow_{\text{L}} c_1 \in claves \wedge_{\text{L}} c \in conectadas(conexiones(r), c_1)$

2.4 Función de abstracción

Abs : $\widehat{\texttt{redstr}}\ rstr \longrightarrow \widehat{\texttt{red}}$ $\{\text{Rep}(rstr)\}$

$(\forall rstr : \widehat{\texttt{redstr}})$
$\text{Abs}(rstr) \equiv r : \widehat{\texttt{red}}\ |$
$computadoras(r) =_{\text{obs}} rstr.compus \wedge$
$(\forall c_0, c_1 : compu)\ conectadas?(r, c_0, c_1) =_{\text{obs}} c_1 \in conectadas(rstr.conexiones, c_0) \wedge$
$interfazUsada(r, c_0, c_1) =_{\text{obs}} dameClave(significado(rstr.conexiones, c_0), c_1)$

Extensión TAD diccionario

$conectadas : dicc(compu, dicc(interfaz, compu))\ d \times compu\ c \to conj(compu)$

$conectadas(d) = obtenerSignificados(significado(d, c), claves(significado(d, c)))$

$obtenerSignificados : dicc(interfaz, compu)\ d \times conj(interfaz)\ ci \to conj(compu)$

$obtenerSignificados(d, c) =$
if $(\#(c) = 0)$ **then**
$\emptyset$
else
$significado(d, dameUno(c)) \cup obtenerSignificado(d, sinUno(c))$
fi

2.5 Algoritmos

iNueva() $\longrightarrow$ *res* : `redstr`

$res \leftarrow CrearTupla(compus : conj(compu), conexiones : dicc(compu, dicc(interfaz, compu)))$	O(1)
$res.conexiones \leftarrow Vacio()$	O(1)
$res.compus \leftarrow Vacio()$	O(1)
return res	
	O(1)

iAgregarCompu(**in/out** r : `redstr`, *in* c : `compu`)

$AgregarRapido(r.compus, c)$	O(1)
	O(1)

iConectar(**in/out** r : `redstr`, *in* c_0 : `compu`, *in* c_1 : `compu`, *in* i_0 : `interfaz`, *in* i_1 : `interfaz`)

if $\neg$ ($definido?(r.conexiones, c_0)$) **then**	
$definir(r.conexiones, c_0, Vacio())$	O(#(*r.compus*))
end if	
if $\neg$ ($definido?(r.conexiones, c_1)$) **then**	
$definir(r.conexiones, c_1, Vacio())$	O(#(*r.compus*))
end if	
$definir(significado(r.conexiones, c_0), i_0, c_1)$	O(#(*r.compus*))
$definir(significado(r.conexiones, c_1), i_1, c_0)$	O(#(*r.compus*))
	O(# (*r.compus*))

iStringAIndice(**in** r : `redstr`, *in* c : `compu`) $\longrightarrow$ *res* : `nat`

$itCompus \leftarrow CrearIt(r.compus)$	O(1)
// $L = longituddeIPmaslarga$	
while $itCompus.siguiente().IP! = c.IP$ **do**	O(L)
$res++$	O(1)
$avanzar(itCompus)$	O(1)
end while	
return res	O(1)
	O(#(*r.compus*) $*$ L)

iUsaInterfaz(**in** r : `redstr`, *in* c : `compu`, *in* i : `interfaz`) $\longrightarrow$ *res* : `bool`

if $definido?(r.conexiones, c)$ **then**	O(#(*r.compus*))
res $\leftarrow$ $definido?(significado(r.conexiones, c), i)$	O(#(*r.compus*))
else	
res $\leftarrow$ $false$	O(1)
end if	
return res	O(1)
	O(#(*r.compus*))

iConectadas?(**in** r : `redstr`, *in* c_1 : `compu`, *in* c_2 : `compu`) $\longrightarrow$ *res* : `bool`

$res \leftarrow pertence(c_2, significados(significado(r.conexiones, c_1)))$	O(#(*r.compus*))
return res	O(1)
	O(#(*r.compus*))

iInterfazUsada(**in** r : `redstr`, *in* c_1 : `compu`, *in* c_2 : `compu`) $\longrightarrow$ *res* : `interfaz`

$res \leftarrow dameClave(significado(d, c_1), c_2)$	O(#(*r.compus*))
return res	O(1)
	O(#(*r.compus*))

iCaminoMinimo(**in** r : `red`, *in* f : `compu`, *in* d : `compu`) $\longrightarrow$ *res* : `bool`

var $auxCaminos$: $Lista(Lista(compu)) \leftarrow Vacia()$	O(1)
var $visitados$: $Lista(compu) \leftarrow Vacia()$	O(1)
AgregarAdelante(auxCaminos, Lista(f))	O(1)
AgregarAdelante(visitados, f)	O(1)
while $Longitud(auxCaminos)! = 0$ **do**	O(1)

var $camino : Lista(compu) \leftarrow sacarMenor(auxCaminos)$	
	$O(n*n)$
if $Ultimo(camino) == d$ **then**	n
$res \leftarrow camino$	$O(1)$
end if	
var $vecinos : Lista(compu) \leftarrow nuevosVecinos(camino, visitados)$	
	$O(n*n*n)$
while $Longitud(vecinos)! = 0$ **do**	
var $caminoAux : Lista(compu) \leftarrow camino$	$O(1)$
$AgregarPrimero(caminoAux, vecinos[0])$	$O(1)$
$Eliminar(vecinos, vecinos[0])$	$O(n)$
$AgregarPrimero(auxCaminos, caminoAuxiliar)$	$O(n*n)$
end while	
$AgregarAVisitados(visitados, auxCaminos)$	$O(n*n)$
end while	
$res \leftarrow Vacia(Vacia())$	$O(1)$
	$O(n^5)$

iNuevosVecinos(**in** r : `redstr`, *in* $camino : Lista(compu)$, *in* $visitados : Lista(compu)) \longrightarrow res : Lista(compu)$

var $long : nat \leftarrow Longitud(camino)$	$O(1)$
var $vecinos : Lista \leftarrow significados(significado(r.conexiones, camino[long-1]))$	
	$O(1)$
var $itVecinos : itDicc(interfaz, compu) \leftarrow CrearIterador(vecinos)$	
	$O(1)$
var $haySiguiente$: *bool* $\leftarrow (longitud(vecinos)! = 0)$	$O(1)$
while $haySiguiente$ **do**	$O(\#(r.compus))$
var $vecino$: $compu \leftarrow actual(itVecinos)$	$O(1)$
if $\neg(Esta(visitados, vecino))$ **then**	$O(1)$
$AgregarUltimo(vecinos, vecino)$	$O(1)$
end if	
$avanzar(itVecinos)$	$O(1)$
$haySiguiente \leftarrow haySiguiente(itVecinos)$	$O(1)$
end while	
$res \leftarrow vecinos$	
	$O(\#(r.compus))$

iSacarMenor(**in/out** $caminos : Lista(Lista(compu)) \longrightarrow res : Lista(compu)$

var $size : nat \leftarrow Longitud(caminos)$	
var $i : nat \leftarrow 0$	

```
    while i < size do

        if Longitud(caminos[i]) == minimo(caminos) then
            Eliminar(caminos, caminos[i])
            res ← caminos[i]
        end if
        i++
    end while
```

iAgregarVisitados(**in/out** *visitados* : *Lista*(*compu*), *in caminos* : *Lista*(*Lista*(*compu*)))

```
    var size : nat ← Longitud(caminos)

    var i : nat ← 0

    while i < size do
        AgregarUltimo(visitados, Ultimo(caminos[i]))
        i++
    end while
```

3 ConjLog

3.1 Interfaz($\alpha, =_\alpha, <_\alpha$)

3.1.1 parámetros formales

géneros α

operaciones

- $=_\alpha \bullet : \alpha \times \alpha \rightarrow bool$ Relación de equivalencia
- $<_\alpha \bullet : \alpha \times \alpha \rightarrow bool$ Relación de orden

se explica con CONJ(α)

géneros `conjLog(`α`)`

Operaciones

NUEVO() $\longrightarrow$ *res* : `conjLog(`α`)`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \emptyset\}$
Descripción: Crea un nuevo conjLog vacio
Complejidad: O(1)

VACÍO?(**in** *cl* : `conjLog(`α`)`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv \{res = (\emptyset?(cl))\}$
Descripción: Indica si el conjunto es vacío
Complejidad: $O(1)$

PERTENECE(**in** *cl* : `conjLog(`α`)`, **in** $e : \alpha$) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv \{res = (e \in cl)\}$
Descripción: Retorna un booleano que indica si el elemento pertenece al conjunto
Complejidad: $O(log(\#(cl)))$

BUSCAR(**in** *cl* : `conjLog(`α`)`, **in** $e : \alpha$) $\longrightarrow$ *res* : α
Pre $\equiv \{e \in cl\}$
Post $\equiv \{res == e\}$
Descripción: Devuelve el elemento que se está buscando
Complejidad: $O(log(\#(cl)))$
Aliasing: El elemento se devuelve por referencia y es modificable

MENOR(**in** *cl* : `conjLog(`α`)`, *in* $e : \alpha$) $\longrightarrow$ *res* : α
Pre $\equiv \{e \in cl\}$
Post $\equiv \{res == max(cl)\}$
Descripción: Devuelve el menor elemento del conjunto
Complejidad: $O(log(\#(cl)))$
Aliasing: El elemento se devuelve por referencia y es modificable

INSERTAR(**in/out** *cl* : `cl(`α`)`, *in* $e : \alpha$)
Pre $\equiv \{cl_0 =_{\text{obs}} cl \wedge \neg(e \in cl)\}$
Post $\equiv \{cl_0 =_{\text{obs}} Agregar(cl_0, e)\}$

Descripción: Inserta un nuevo elemento en el conjunto
Complejidad: O($log(\#(cl))$)

BORRAR(**in/out** cl : `cl(`α`)`, $in\ e : \alpha$)
Pre $\equiv \{cl_0 =_{\text{obs}} cl \wedge (e \in cl)\}$
Post $\equiv \{cl =_{\text{obs}} (cl_0 - \{e\})\}$
Descripción: Elimina el elemento e del conjunto cl, los iteradores que apunten a este elemento se indefinen
Complejidad: O($log(\#(cl))$)

3.2 Representación

se representa con `clog`

donde `clog` **es** `raiz : puntero(nodo)`
donde `nodo` **es** `tupla`⟨der : `puntero(nodo)`,
izq : `puntero(nodo)`,
valor : α,
padre : `puntero(nodo)`,
fdb : `nat`⟩

3.3 Invariante de representación

1. Para todas las raíces, la altura del subárbol derecho menos la altura del subárbol izquierdo de esa raíz es igual al fdb.
2. El fdb de todas las raíces es 0, 1 o -1.
3. Si un nodo no es una hoja del árbol entonces los padres de los hijos derecho e izquierdo son iguales y es el nodo
4. Si un nodo es una hoja del arbol entonces los hijos derecho e izquierdo del árbol son NULL
5. Para todos los nodos n, todos los nodos del subárbol derecho son mayores que n
6. Para todos los nodos n, todos los nodos del subárbol izquierdo son menores que n
7. No hay nodos repetidos
8. El padre de la raíz es NULL

3.4 Función de abstracción

$$\text{Abs} : \widehat{\texttt{clog(}\alpha\texttt{)}}\ cl \longrightarrow \widehat{\texttt{conj(}\alpha\texttt{)}} \qquad \{\text{Rep}(cl)\}$$

$(\forall cl : \widehat{\texttt{clog(}\alpha\texttt{)}})$
$\text{Abs}(cl) \equiv c : \widehat{\texttt{conj(}\alpha\texttt{)}} \mid$
$((\forall e : \alpha) e \in c \ \Rightarrow_{\text{L}}\ esta(cl, e)) \wedge size(cl) = \#(c)$

3.4.1 Aclaración de complejidades

Este módulo es un AVL y fue diseñado para poder cumplir con las especificaciones de complejidad de dcnet. Si bien es paramétrico, está pensado para trabajar con tipo 'paquete' en el que la copia se puede realizar en tiempo constante. En resumen, todas las operaciones de comparación, asignacion y copia del tipo paramétrico α deben poder ser realizadas en O(1).

3.5 Algoritmos

```
iVacío?(in cl : conjLog(α)) ⟶ res : bool                                  O(1)
  res ← cl == NULL                                                        ──────
                                                                          O(1)

iBorrar(in/out cl : conjLog(α),  in e : α)

  var variandoHijoDerecho? : bool ← true

  var clactual : conjLog(α) ← cl                                          O(1)

  var aBorrar : conjLog(α)

  if (¬(cl.der == NULL) ∧ ¬(cl.izq == NULL)) then                         O(1)
    clactual ← iEncontrarPadre(clactual, e)                               O(log(size(cl)))
    if cactual.der! = NULL ∧_L cactual.der.valor == e then
      aBorrar ← cactual.der                                               O(1)
    else
      aBorrar ← cactual.izq                                               O(1)
    end if

    var mm : conjLog(α) ← iDameMayorMenores(clactual)
                                                                          O(log(size(cl)))

    if mm.valor == e then                                                 O(1)

      if mm.padre.der! = NULL ∧_L mm.padre.der.valor == mm.valor then
        variandoHijoDerecho? ← true                                       O(1)
        mm.padre.der = NULL
        mm.padre.fdb − −
      else
        variandoHijoDerecho? ← false                                      O(1)
        mm.padre.izq = NULL
        mm.padre.fdb + +
      end if
    else

      var mmValor : α ← mm.valor                                          O(1)
      aBorrar.valor ← mmValor                                             O(1)

      if mm.izq! = NULL then                                              O(1)
        mm.valor ← mm.izq.valor                                           O(1)
        mm.izq ← NULL                                                     O(1)
        mm.fdb + +                                                        O(1)

        if mm.padre.valor == e then
          variandoHijoDerecho? ← false
        else
          variandoHijoDerecho? ← true
        end if
      else
```

```
            if mm.padre.valor == e then                                   O(1)
                mm.padre.izq = NULL                                       O(1)
                variandoHijoDerecho? ← false                              O(1)
            else
                mm.padre.der = NULL                                       O(1)
                variandoHijoDerecho? ← true                               O(1)
            end if
        end if
    end if
    iRebYRecalcFDB(mm.padre, variandoHijoDerecho?, estoyBorrando?)
                                                                          O(log(size(cl)))
else

    if cl.der == NULL ∧ cl.izq == NULL then                               O(1)
        cl ← NULL                                                         O(1)
    else

        if cl.der == NULL then                                            O(1)

            if cl.izq.valor == e then                                     O(1)
                cl.izq ← NULL                                             O(1)
            else
                cl.valor ← cl.izq.valor                                   O(1)
                cl.izq ← NULL                                             O(1)
            end if
        else

            if cl.der.valor == e then                                     O(1)
                cl.der ← NULL                                             O(1)
            else
                cl.valor ← cl.der.valor                                   O(1)
                cl.der ← NULL                                             O(1)
            end if
        end if
    end if
end if
                                                                          ---------
                                                                          O(log(size(cl)))
```

Justificación de complejidad

Por álgebra de órdenes

```
iInsertar(in/out cl : conjLog(α),  in e : α)

    var clactual : conjLog(α) ← cl                                        O(1)

    if ¬(cl.der == NULL) ∧ ¬(cl.izq == NULL) then                         O(1)
        clactual ← iEncontrarPadre(clactual, e)                           O(log(size(cl)))

        if clactual.valor < e then
```

```
        clactual.der ←tupla⟨der : NULL,                                   O(1)
                            izq : NULL,
                            valor : e,
                            padre : clactual,
                            fdb : 0⟩
        iRebYRecalcFDB(clactual, true, false)
    else
        clactual.izq ←tupla⟨der : NULL,                                   O(1)
                            izq : NULL,
                            valor : e,
                            padre : clactual,
                            fdb : 0⟩
        iRebYRecalcFDB(clactual, false, false)
    end if
else

    if cl.der == NULL ∧ cl.izq == NULL then                               O(1)
        cl ←tupla⟨der : NULL,                                             O(1)
                  izq : NULL,
                  valor : e,
                  padre : clactual,
                  fdb : 0⟩
    else

        if cl.der! = NULL then
            cl.izq ←tupla⟨der : NULL,                                     O(1)
                        izq : NULL,
                        valor : e,
                        padre : cl,
                        fdb : 0⟩
        else
            cl.der ←tupla⟨der : NULL,                                     O(1)
                        izq : NULL,
                        valor : e,
                        padre : cl,
                        fdb : 0⟩
        end if
    end if
end if
```

O($log(size(cl))$)

Justificación de complejidad

Por álgebra de órdenes

iPertenece(**in/out** cl : `conjLog(`α`)`, *in* $e : \alpha$) $\longrightarrow$ res : `bool`

```
var encontrado? : bool ← false                                            O(1)

var clactual : conjLog(α) ← cl                                            O(1)

while (clactual! = NULL) ∧ ¬(encontrado?) do                              O(1)
```

if $e > clactual.valor$ **then**	O(1)
$clactual \leftarrow clactual.der$	O(1)
else	
if $ce < clactual.valor$ **then**	O(1)
$clactual \leftarrow clactual.izq$	O(1)
else	
$encontrado? \leftarrow true$	O(1)
end if	
end if	
end while	
$clactual \leftarrow NULL$	O(1)
$res \leftarrow encontrado?$	O(1)
	$O(log(size(cl)))$

Justificación de complejidad

El ciclo recorre a lo sumo una rama del árbol (el árbol está ordenado), teniendo ésta como máximo la longitud del árbol (sus alturas no difieren en más de una hoja) que es log(n), con n = size(cl)

iMenor(**in** cl : `conjLog(`α`)`) $\longrightarrow$ $res : \alpha$

var $clactual$: `conjLog(`α`)` $\leftarrow cl$	O(1)
$clactual \leftarrow iMenorNodo(clactual)$	$O(log(size(cl)))$
$res \leftarrow clactual.valor$	O(1)
	$O(log(size(cl)))$

Justificación de complejidad

Por álgebra de órdenes

iBuscar(**in** cl : `conjLog(`α`)`, $e : \alpha$) $\longrightarrow$ $res : \alpha$

var $padre$: `conjLog(`α`)` $\leftarrow iEncontrarPadre(cl, e)$	$O(log(size(cl)))$
if $padre.der! = NULL \wedge_{\mathrm{L}} padre.der.valor == e$ **then**	O(1)
$res \leftarrow padre.der.valor$	
else	
$res \leftarrow padre.izq.valor$	
end if	
	O($log(size(cl))$)

Justificación de complejidad

Por álgebra de órdenes

3.6 Auxiliares

iRebYRecalcFDB(**in/out** cl : `conjLog(`α`)`, *in variandoHijoDerecho?* : `bool`, *in estoyBorrando?* : `bool`)

```
var clactual : conjLog(α) ← cl                                   O(1)

var termino? : bool ← false                                      O(1)

if estoyBorrando? then

    while clactual! = NULL ∧ ¬(termino?) do                      O(1)

        if variandoHijoDerecho? then

            if clactual.fdb == −1 then                           O(1)

                var fdbIzq : nat                                 O(1)

                if cl.izq! = NULL then
                    fdbIzq ← cl.izq                              O(1)
                end if

                if cl.izq! = NULL ∧_L cl.izq.fdb == 1 then       O(1)
                    iRotarLR(cl.izq)                             O(1)
                end if
                iRotarLL(cl)                                     O(1)

                if cl.izq! = NULL ∧_L fdbIzq == 0 then           O(1)
                    termino? ← true                              O(1)
                end if
            else

                if cl.fdb == +1 then                             O(1)
                    cl.fdb ← 0                                   O(1)
                    termino? ← true                              O(1)
                else
                    cl.fdb ← −1                                  O(1)
                end if
            end if
        else

            if clactual.fdb == −1 then                           O(1)

                var fdbDer : nat                                 O(1)

                if cl.der! = NULL then                           O(1)
                    fdbDer ← cl.der.fdb                          O(1)
                end if

                if cl.der! = NULL ∧_L fdbDer == 1 then           O(1)
                    iRotarRL(cl.der)                             O(1)
```

```
            end if
            iRotarRR(cl)                                                    O(1)

            if fdbDer == 0 then                                             O(1)
                termino? ← true                                             O(1)
            end if
        else

            if cl.fdb == −1 then                                            O(1)
                cl.fdb ← 0                                                  O(1)
                termino? ← true                                             O(1)
            else
                cl.fdb ← +1                                                 O(1)
            end if
        end if
    end if
    variandoHijoDerecho ← (cl.padre! = NULL ∧L cl.padre.der.valor == cl.valor)
                                                                            O(1)
    clactual ← clactual.padre                                               O(1)
  end while
else    // No hubo borrado, entonces hubo una inserción

  while clactual! = NULL ∧ ¬(termino?) do                                   O(1)

    if variandoHijoDerecho? then

      if clactual.fdb == +1 then                                            O(1)

        var fdbDer : nat                                                    O(1)

        if cl.der! = NULL then
            fdbDer ← cl.der.fdb                                             O(1)
        end if

        if cl.der! = NULL ∧L fdbDer == −1 then                              O(1)
            iRotarRL(cl.der)                                                O(1)
        end if
        iRotarRR(cl)                                                        O(1)
        termino? ← true
      else

        if clactual.fdb == −1 then                                          O(1)
            clactual.fdb ← 0                                                O(1)
            termino? ← true                                                 O(1)
        else
            clactual.fdb ← 1                                                O(1)
        end if
      end if
    else

      if clactual.fdb == −1 then                                            O(1)
        fdbIzq : nat                                                        O(1)
```

```
            if cl.izq! = NULL then
                fdbIzq ← cl.izq.fdb                                   O(1)
            end if

            if cl.izq! = NULL ∧_L fdbIzq == +1 then                   O(1)
                iRotarLR(cl.izq)                                      O(1)
            end if
            iRotarLL(cl)                                              O(1)
            termino? ← true                                           O(1)
        else

            if clactual.fdb == +1 then                                O(1)
                clactual.fdb ← 0                                      O(1)
                termino? ← true                                       O(1)
            else
                clactual.fdb ← −1                                     O(1)
            end if
        end if
    end if
    variandoHijoDerecho ← (cl.padre! = NULL ∧_L cl.padre.der.valor == cl.valor)
                                                                      O(1)
    clactual ← clactual.padre                                         O(1)
  end while
end if
                                                                      ---
                                                                      O(log(size(cl)))
```

Justificación de complejidad

En éste ciclo, tanto para el borrado y para la inserción se tiene un nodo interno que fue modificado y a partir de éste se comienza a subir hasta llegar como máximo al nodo raíz del árbol,recorriendo como mucho la altura del árbol

iRotarRR(**in/out** *cl* : `conjLog(`α`)`)

```
var nietoRR : conjLog(α) ← cl.der.der                                 O(1)

var hijoDer : conjLog(α) ← cl.der                                     O(1)

var hijoIzq : conjLog(α) ← cl.izq                                     O(1)
cl.der ← NULL                                                         O(1)
cl.izq =tupla⟨der : hijoDer.der,                                      O(1)
            izq : cl.izq,
            valor : cl.valor,
            padre : cl,
            fdb : 0⟩
cl.izq.izq.padre ← cl.izq                                             O(1)
cl.izq.der.padre ← cl.izq                                             O(1)
cl.valor = hijoDer.valor                                              O(1)
cl.der = nietoRR                                                      O(1)
cl.der.padre ← cl                                                     O(1)
                                                                      ---
                                                                      O(1)
```

Justificación de complejidad

Son operaciones sobre α y punteros

iRotarRL(**in/out** cl : `conjLog(`α`)`)

var $nietoRR$: `conjLog(`α`)` $\leftarrow cl.der.der$	O(1)
var $nietoRL$: `conjLog(`α`)` $\leftarrow cl.der.izq$	O(1)
var $valorDer$: $\alpha \leftarrow cl.der.valor$	O(1)
$cl.der.valor \leftarrow nietoRL.valor$	O(1)
$nietoRR.izq \leftarrow nietoRL.der$	O(1)
$cl.der.der \leftarrow nietoRR$	O(1)
$cl.der.izq \leftarrow nietoRL.izq$	O(1)
$cl.der.der.padre \leftarrow cl.der$	O(1)
$cl.der.izq.padre \leftarrow cl.der$	O(1)
$cl.izq.fdb \leftarrow +1$	
	$O(1)$

Justificación de complejidad

Son operaciones sobre α y punteros

iRotarLL(**in/out** cl : `conjLog(`α`)`)

var $nietoLL$: `conjLog(`α`)` $\leftarrow cl.izq.izq$	O(1)
var $hijoIzq$: `conjLog(`α`)` $\leftarrow cl.izq$	O(1)
var $hijoDer$: `conjLog(`α`)` $\leftarrow cl.der$	O(1)
$cl.izq \leftarrow NULL$	O(1)
$cl.der =$`tupla`⟨der : `cl.der`, izq : `hijoIzq.der`, valor : `cl.valor`, padre : `cl`, fdb : `0`⟩	O(1)
$cl.der.izq.padre \leftarrow cl.der$	O(1)
$cl.der.der.padre \leftarrow cl.der$	O(1)
$cl.valor = hijoIzq.valor$	O(1)
$cl.izq = nietoLL$	O(1)
$cl.izq.padre \leftarrow cl$	O(1)
	$O(1)$

Justificación de complejidad

Son operaciones sobre α y punteros

iRotarLR(**in/out** cl : `conjLog(`α`)`)

var $nietoLL$: `conjLog(`α`)` $\leftarrow cl.izq.izq$	O(1)

var $nietoLR$: `conjLog(`α`)` $\leftarrow cl.izq.der$	O(1)
var $valorIzq : \alpha \leftarrow cl.izq.valor$	O(1)
$cl.izq.valor \leftarrow nietoRL.valor$	O(1)
$nietoLL.izq \leftarrow nietoLR.izq$	O(1)
$cl.izq.izq \leftarrow nietoLL$	O(1)
$cl.izq.der \leftarrow nietoLR.der$	O(1)
$cl.izq.izq.padre \leftarrow cl.izq$	O(1)
$cl.izq.der.padre \leftarrow cl.izq$	O(1)
$cl.izq.fdb \leftarrow -1$	O(1)
	$O(1)$

Justificación de complejidad

Son operaciones sobre α y punteros

iEncontrarPadre(**in** cl : `conjLog(`α`)`, $e : \alpha$) $\longrightarrow$ res : `conjLog(`α`)`

var $clactual$: `conjLog(`α`)`	O(1)
var $encontrado?$: `bool` $\leftarrow (clactual.der! = NULL \wedge_{\mathrm{L}} clactual.der.valor == e) \vee (clactual.izq! = NULL \wedge_{\mathrm{L}} clactual.izq.valor == e)$	
while $\neg encontrado?$ **do**	
if $e > clactual.valor$ **then**	
$clactual \leftarrow clactual.der$	$O(1)$
else	
$clactual \leftarrow clactual.izq$	$O(1)$
end if	
$encontrado? \leftarrow (clactual.der! = NULL \wedge_{\mathrm{L}} clactual.der.valor == e) \vee (clactual.izq! = NULL \wedge_{\mathrm{L}} clactual.izq.valor == e)$	
end while	
$res \leftarrow clactual$	$O(1)$
	$O(size(cl))$

Justificación de complejidad

El ciclo recorre a lo sumo una rama del árbol (el árbol está ordenado), teniendo ésta como máximo la altura del árbol (sus alturas no difieren en más de una hoja) que es log(n), con n = size(cl)

iDameMayorMenores(**in** cl : `conjLog(`α`)`, $e : \alpha$) $\longrightarrow$ res : `conjLog(`α`)`

var $clactual$: `conjLog(`α`)` $\leftarrow cl$	O(1)
if $clactual.izq! = NULL$ **then**	O(1)
$clactual \leftarrow iMayorNodo(clactual)$	$O(log(size(cl)))$
end if	
$res \leftarrow clactual$	O(1)

Justificación de complejidad

Por álgebra de órdenes

iMenorNodo(**in** cl : conjLog(α)) $\longrightarrow$ res : $conjLog(\alpha)$

var $clactual$: conjLog(α) $\leftarrow cl$	O(1)
while $clactual.izq! = NULL$ **do**	
$clactual \leftarrow clactual.izq$	
end while	
$res \leftarrow clactual$	O(1)
	$O(log(size(cl)))$

Justificación de complejidad

Para encontrar el menor nodo se recorre el árbol siempre a la izquierda, se alcanza a recorrer una única rama, es decir la altura del árbol

iMayorNodo(**in** cl : conjLog(α)) $\longrightarrow$ res : $conjLog(\alpha)$

var $clactual$: conjLog(α) $\leftarrow cl$	O(1)
while $clactual.der! = NULL$ **do**	
$clactual \leftarrow clactual.der$	
end while	
$res \leftarrow clactual$	O(1)
	$O(log(size(cl)))$

Justificación de complejidad

Para encontrar el mayor nodo se recorre el árbol siempre a la derecha, se alcanza a recorrer una única rama, es decir la altura del árbol

size(**in** cl : conjLog(α)) $\longrightarrow$ res : nat

if $cl == NULL$ **then**
  $res \leftarrow 0$
else
  $res \leftarrow 1 + iSize(cl.der) + iSize(cl.izq)$
end if

3.7 Operaciones auxiliares de conj(α)

$menor : conj(\alpha)\ c \rightarrow \alpha \qquad \{\#(c) > 0\}$

$menor(c) =$
  if $(\#(c) = 1)$ **then**
    $dameUno(c)$
  else
    if $(dameUno(c) < menor(sinUno(c))$ **then**

$dameUno(c)$

else

$menor(sinUno(c))$

fi

fi

4 Diccionario por Prefijos

El módulo Diccionario por prefijos provee un diccionario en el que las claves son secuencias no acotadas de caracteres. Con el se puede definir una clave, obtener un significado y eliminar una clave. Estas tres operaciones están definidas en tiempo O(L) con L la máxima longitud del conjunto de las claves introducidas (y cuando se está definiendo una clave se incluye en el conjunto la clave a introducir).

4.1 Interfaz

parámetros formales

géneros β

se explica con DICCIONARIO(SECU(CHAR), β)

géneros `diccPref(secu(char)`, β`)`

Operaciones

NUEVO() $\longrightarrow$ *res* : `diccPref(secu(char),`β`)`
Pre $\equiv$ {true}
Post $\equiv$ {(∀ p:secu(char)) ¬(def?(p,*res*))}
Descripción: Crea un nuevo diccionario vacío
Complejidad: O(1)

DEF?(**in** *dp* : `diccPref(secu(char),`β`)`, *in p* : `secu(char)`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res=p ∈ claves(dp)}
Descripción: Devuelve true o false según si la clave está o no definida
Complejidad: O(L)

CLAVES(**in** *dp* : `diccPref(secu(char),`β`)`) $\longrightarrow$ *res* : `conj(secu(char))`
Pre $\equiv$ {true}
Post $\equiv$ {(∀ c: secu(char))c ∈ claves(dp) $\iff$ def?(c,dp)}
Descripción: Devuelve un conjunto de las claves del diccionario
Complejidad: O(n)

DEFINIR(**in/out** *dp* : `diccPref(secu(char),`β`)`, *in p* : `secu(char)`, *in s* : β)
Pre $\equiv$ {dp=dp_0 ∧ ¬def?(p,dp)}
Post $\equiv$ {def?(p,dp) ∧ obtener(p,dp)=$_{\text{obs}}$s ∧ (∀ c ∈ claves(dp_0)) def?(c,dp)}
Descripción: Inserta una nueva clave con su significado en el diccionario
Complejidad: O(L)

OBTENER(**in** *dp* : `diccPref(secu(char),`β`)`, **in** *p* : `secu(char)`) $\longrightarrow$ *res* : β
Pre $\equiv$ {def?(p,dp)}
Post $\equiv$ {$res = obtener(p, dp)$}
Descripción: Retorna el significado de la clave pedida
Complejidad: O(L)
Aliasing: Devuelve res por referencia

ELIMINAR(**in/out** *dp* : `diccPref(secu(char),`β`)`, **in** *p* : `secu(char)`)
Pre $\equiv$ {dp=dp_0 ∧ def?(p,dp)}
Post $\equiv$ {¬def?(dp) ∧(∀$c \in claves(dp_0), c \neq$ p)def?(c,dp)}

Descripción: Elimina del diccionario la clave deseada
Complejidad: O(L)

5 Paquete

Un Paquete representa a un paquete a partir de una tupla que contiene el id del paquete, la prioridad, el origen el destino y un indicador de en que parte de su camino está.

5.1 Interfaz

se explica con PAQUETE

géneros `paquete`

Operaciones

CREARPAQUETE(**in** id : `nat`, **in** o : `compu`, **in** d : `compu`, **in** pr : `nat`) $\longrightarrow$ res : `paquete`
Pre $\equiv$ {true}
Post $\equiv$ $\{pi_1(res) = id \wedge pi_2(res) = pr \wedge pi_3(res) = o \wedge pi_4(res) = d\}$
Descripción: Crea un paquete
Complejidad: O(1)

$\bullet <_p \bullet$(**in** p_1 : `paquete`, *in* p_2 : `paquete`) $\longrightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res=true $\iff$ $(\pi_2(p_1) = \pi_2(p_2) \wedge pi_1(p_1) < pi_1(p_2) \vee (\pi_1(p_1) < \pi_1(p_2)))$}
Descripción: Define un orden en paquete según la prioridad
Complejidad: O(1)

$\bullet <_{id} \bullet$(**in** p_1 : `paquete`, *in* p_2 : `paquete`) $\longrightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res=true $\iff$ $\mathrm{id}(\mathrm{p}_1)<\mathrm{id}(\mathrm{p}_2)$}
Descripción: Define un orden en paquete según el id
Complejidad: O(1)

ID(**in** p : `paquete`) $\longrightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = \pi_1(paquete)\}$
Descripción: *Getterdeid*
Complejidad: O(1)

PRIORIDAD(**in** p : `paquete`) $\longrightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = \pi_2(paquete)\}$
Descripción: *Getterdeprioridad*
Complejidad: O(1)

ORIGEN(**in** p : `paquete`) $\longrightarrow$ res : `Ip`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = \pi_3(paquete)\}$
Descripción: *Getterdeorigen*
Complejidad: O(1)

DESTINO(**in** p : `paquete`) $\longrightarrow$ res : `Ip`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = \pi_4(paquete)\}$
Descripción: *Getterdedestino*
Complejidad: O(1)

5.2 Representación

se representa con `paquete`

donde `paquete` **es** `tupla`⟨id : `nat`,
origen : `Ip`,
destino : `Ip`,
prioridad : `nat`⟩

6 PaquetePos

Un PaquetePos es

6.1 Interfaz

se explica con `tupla`⟨ : `Paquete`,
: `nat`,
: `nat`,
: `nat`⟩

géneros `paquetePos`

Operaciones

CREARPAQUETE(**in** id : `nat`, **in** o : `compu`, **in** d : `compu`, **in** pr : `nat`) $\longrightarrow$ res : $< paquete, nat, nat, nat >$
Pre $\equiv$ {true}
Post $\equiv$ $\{pi_1(pi_1(res)) = id \wedge pi_2(pi_1(res)) = pr \wedge pi_3(pi_1(res)) = o \wedge pi_4(pi_1(res)) = d\}$
Descripción: Crea un paquete
Complejidad: O(1)

• $<_p$ •(**in** p_1 : $< paquete, nat, nat, nat >$, **in** p_2 : $< paquete, nat, nat, nat >$) $\longrightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res=true $\iff$ $(\pi_2(\pi_1(p_1)) = \pi_2(\pi_1(p_2)) \wedge \pi_1(\pi_1((p_1)) < \pi_1(\pi_1((p_2)) \vee (\pi_1(\pi_1((p_1)) < \pi_1(\pi_1((p_2))))$}
Descripción: Define un orden en paquete según la prioridad
Complejidad: O(1)

• $<_{id}$ •(**in** p_1 : $< paquete, nat, nat, nat >$, *in* p_2 : $< paquete, nat, nat, nat >$) $\longrightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res=true $\iff$ $(pi_1(pi_1(p_1)) < (pi_1(pi_1(p_2))))$}
Descripción: Define un orden en paquete según el id
Complejidad: O(1)

GETPAQUETE(**in** $ppos$: $< paquete, nat, nat, nat >$) $\longrightarrow$ res : `paquete`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \pi_1(paquete)\}$
Descripción: Getter de paquete
Complejidad: O(1)

INDICEORIGEN(**in** p : $< paquete, nat, nat, nat >$) $\longrightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = \pi_2(p)\}$
Descripción: Getter de indiceOrigen
Complejidad: O(1)

INDICEDESTINO(**in** p : $< paquete, nat, nat, nat >$) $\longrightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = \pi_3(p)\}$
Descripción: Getter de indiceDestino
Complejidad: O(1)

POSACTUAL(**in** p : $< paquete, nat, nat, nat >$) $\longrightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res = \pi_4(p)\}$

Descripción: Getter de posActual
Complejidad: O(1)

ACTUALIZARPOSACTUAL(**in/out** $p : < paquete, nat, nat, nat >$)
Pre $\equiv \{p_1 =_{\text{obs}} p\}$
Post $\equiv \{posActual(p) = posActual(p_1) + 1\}$
Descripción: Aumentar la posicion actual
Complejidad: O(1)

6.2 Representación

se representa con `paqPos`

donde `paqPos` **es** `tupla`⟨paquete : `paquete`,
indiceOrigen : `nat`,
indiceDestino : `nat`,
posActual : `nat`⟩

Justificación de estructura

Las restricciones de complejidad del tipo dcnet para caminoRecorrido y enEspera nos obligaron a tener representadas dos estructuras distintas para almacenar los paquetes. En una de ellas tenemos los paquetes a retornar por la operación enEspera que devuelve los paquetes en la cola de cierta computadora y en la otra devolvemos una estructura similar a paquete (paquetePos) con información adicional acerca de las posiciones en que se encuentra el paquete que nos permite encontrar el caminoRecorrido en el tiempo solicitado.