



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, Jérica | 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 | 2 |
| 1.1. Interfaz | 2 |
| 1.2. Representación | 3 |
| 1.3. Servicios Usados | 7 |
| 2. Módulo Red | 8 |
| 2.1. Interfaz | 8 |
| 2.2. Extensión del TAD Red | 9 |
| 2.3. Representación | 9 |
| 3. ConjLog | 11 |
| 3.1. Interfaz($\alpha, =_{\alpha}, <_{\alpha}$) | 11 |
| 3.1.1. parámetros formales | 11 |
| 3.2. Representación | 12 |
| 3.3. Invariante de representación | 13 |
| 3.4. Función de abstracción | 13 |
| 3.4.1. Aclaración de complejidades | 13 |
| 3.5. Algoritmos | 14 |
| 3.6. Auxiliares | 18 |
| 3.7. Operaciones auxiliares de $\text{conj}(\alpha)$ | 23 |
| 4. Diccionario por Prefijos | 25 |
| 4.1. Interfaz | 25 |
| 5. Paquete | 27 |
| 5.1. Interfaz | 27 |
| 5.2. Representación | 28 |
| 6. PaquetePos | 29 |
| 6.1. Interfaz | 29 |
| 6.2. Representación | 30 |

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, dicePref, conjLog, conjLogP

géneros dcnet

Operaciones

CREARSISTEMA(**in** $r : \text{red}$) $\longrightarrow res : \text{dcnet}$

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

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

Descripción: Crea un sistema DCNet.

Complejidad: $O(???)$

CREARPAQUETE(**in/out** $s : \text{dcnet}$, **in** $p : \text{paquete}$)

Pre $\equiv \{s =_{\text{obs}} s_0 \wedge (\forall p_0 : \text{paquete}, \text{paqueteEnTransito?}(p, s)) \neg(p_0 =_{\text{obs}} p) \wedge$
 $\text{destino}(p) \in \text{compus}(\text{red}) \wedge \text{origen}(p) \in \text{compus}(\text{red}) \wedge_L$
 $\text{haycamino?}(\text{destino}(p), \text{origen}(p), \text{red}(s))\}$

Post $\equiv \{s =_{\text{obs}} \text{crearPaquete}(s_0, p)\}$

Descripción: Crea un paquete y lo agrega a la computadora correspondiente.

Complejidad: $O(L + \log(k))$ AVANZARSEGUNDO(**in/out** $s : \text{dcnet}$)

Pre $\equiv \{s =_{\text{obs}} s_0\}$

Post $\equiv \{s =_{\text{obs}} \text{avanzarSegundo}(s_0)\}$

Descripción: Avanza un segundo el sistema. Todas las computadoras envían su respectivo paquete y en consecuencia se actualizan los paquetes en espera de cada una de ellas.

Complejidad: $O(n \times (L + \log(n) + \log(k)))$ DAMERED(**in** $s : \text{dcnet}$) $\longrightarrow res : \text{puntero}(\text{red})$

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

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

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** $s : \text{dcnet}$, **in** $p : \text{paquete}$) $\longrightarrow res : \text{secu}(\text{compu})$

Pre $\equiv \{\text{paqueteEnTransito?}(s, p)\}$

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

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** $s : \text{dcnet}$, **in** $c : \text{compu}$) $\longrightarrow res : \text{nat}$

Pre $\equiv \{c \in \text{computadoras}(\text{red}(s))\}$

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

Descripción: Devuelve la cantidad de paquetes enviados por una computadora.

Complejidad: $O(n)$ ENESPERA(**in** $s : \text{dcnet}$, **in** $c : \text{compu}$) $\longrightarrow res : \text{puntero}(\text{conjLogP}(\text{paquete}))$

Pre $\equiv \{c \in \text{computadoras}(\text{red}(s))\}$

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

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 $\text{LAQUEMASENVIO}(\text{in } s : \text{dcnet}) \longrightarrow res : \text{compu}$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{\text{obs}} \text{laQueMasEnvio}(s, p)\}$

Descripción: Devuelve la computadora que más paquetes envió.

Complejidad: $O(1)$ 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 $\text{tupla}(\text{Compus} : \text{arreglo}(\text{tupla}(\text{IP} : \text{String}, \text{pN} : \text{puntero}(\text{conjLog}(\text{paquete})), \text{\#Paquetes} : \text{nat}), \text{CompusPorPref} : \text{diccPref}(\text{compu}, \text{tupla}(\text{PorNom} : \text{conjLog}(\text{paquete}), \text{PorPrior} : \text{conjLog}(\text{paquete})), \text{CaminosMinimos} : \text{arreglo}(\text{arreglo}(\text{arreglo}(\text{compu}))), \text{LaQMaseEnvio} : \text{nat}, \text{Red} : \text{red})$

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 *LaQMaseEnvio*

$\text{Rep} : \widehat{\text{sistema}} \longrightarrow \text{boolean}$

$(\forall s : \widehat{\text{sistema}})$

$\text{Rep}(s) \equiv$

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

Algoritmos

ICREARSISTEMA(**in** $r : \text{red}$) \longrightarrow $res : \text{dcnet}$

```

   $res.red \leftarrow r$ 
   $n \leftarrow \#(\text{COMPUS}(red))$   $O(\#\text{compus}(red)=n)?$ 
   $i \leftarrow 0$ 
   $j \leftarrow 0$   $O(1)$ 
   $res.Compus \leftarrow \text{CREARARREGLO}(n)$   $O(n)$ 
   $res.CaminosMinimos \leftarrow \text{CREARARREGLO}(n)$   $O(n)$ 
  var  $p : \text{arreglo\_dimensionable de puntero}(\text{conjLog}(\text{paquete}))$ 
  while  $i < n$  do  $O(n)$ 
     $res.CaminosMinimos[i] \leftarrow \text{CREARARREGLO}(n)$   $O(n)$ 
     $p[i] \leftarrow \text{NULL}$   $O(1)$ 
     $res.Compus[i] \leftarrow \langle \text{compu}(r, i), p[i], 0 \rangle$   $O(1)$ 
     $s : \langle \text{nat}, \text{conjLog}(\text{paquete}, <_{id}), \text{conjLog}(\text{paquete}, <_p),$ 
     $\text{conjLog}(\langle \text{paquete}, \text{indiceActual}, \text{indiceOrigen}, \text{indiceDestino} \rangle, <_{id}), \text{conjLog}(\langle \text{paquete}, \text{indiceActual}, \text{indiceDestino} \rangle, <_p) \rangle$ 
    HAY que ver bien donde definir la relacion!!
     $\pi_1(s) \leftarrow \text{compu}(r, i)$ 
     $\pi_2(s) \leftarrow \text{NUEVO}()$ 
     $\pi_3(s) \leftarrow \text{NUEVO}()$ 
     $\pi_4(s) \leftarrow \text{NUEVO}()$ 
     $\pi_5(s) \leftarrow \text{NUEVO}()$ 
     $\text{DEFINIR}(res.CompusPorPref, \text{compu}(r, i), s)$   $O(L)$ 
    while  $j < n$  do  $O(n)$ 
       $res.CaminosMinimos[i][j] \leftarrow \text{caminoMinimo}(\text{compu}(r, i), \text{compu}(r, j), r)$ 
       $O(\text{complejidad cammin}(red))$ 
       $j++$ 
    end while
     $i++$ 
  end while
   $res.LaQMasEnvio \leftarrow 0$   $O(1)$ 

```

$O(\max\{n^2 \times O(\text{complejidadcammin}(red)),$

ICREARPAQUETE(**in/out** $s : \text{dcnet}$, **in/out** $p : \text{paquete}$)

```

   $t_1 : \langle \text{nat}, \text{conjLog}(\text{paquete}, <_{id}), \text{conjLog}(\text{paquete}, <_p),$ 
   $\text{conjLog}(\text{paquetePos}, <_{id}), \text{conjLog}(\text{paquetePos}, <_p) \rangle$ 
   $t_1 \leftarrow \text{OBTENER}(\text{origen}(p), s.\text{CompusPorPref})$   $O(L)$ 
   $t_2 : \langle \text{nat}, \text{conjLog}(\text{paquete}, <_{id}), \text{conjLog}(\text{paquete}, <_p),$ 
   $\text{conjLog}(\text{paquetePos}, <_{id}), \text{conjLog}(\text{paquetePos}, <_p) \rangle$ 

```

| | |
|---|-----------------------------|
| $t_2 \leftarrow \text{OBTENER}(\text{destino}(p), s.\text{CompusPorPref})$ | $O(L)$ |
| $p' : \text{paquetePos}$ | |
| $\text{indiceOrigen}(p') \leftarrow \pi_1(t_1)$ | $O(1)$ |
| $\text{indiceDestino}(p') \leftarrow \pi_1(t_2)$ | $O(1)$ |
| $\text{indiceActual}(p') \leftarrow 0$ | |
| $\text{INSERTAR}(\pi_2(t), p)$ | $O(\log(k))$ |
| $\text{INSERTAR}(\pi_3(t), p)$ | $O(\log(k))$ |
| $\text{INSERTAR}(\pi_4(t), p')$ | $O(\log(k))$ |
| $\text{INSERTAR}(\pi_5(t), p')$ | $O(\log(k))$ |
| | <hr/> |
| | $O(L + \log(k))$ |
| ILAQUEMASENVIO (in $s : \text{dcnet}$) $\longrightarrow res : \text{compu}$ | |
| $res \leftarrow s.\text{compus}[s.\text{LaQMasEnvio}].IP$ | $O(1)$ |
| ACLARACION: ACA deberia devolver una compu pero es alto bardo y no puedo hacerlo en $O(1)$ me faltan las interfaces | |
| | <hr/> |
| | $O(1)$ |
| IDAMERED (in $s : \text{dcnet}$) $\longrightarrow res : \text{puntero}(\text{red})$ | |
| $res \leftarrow \&(s.\text{red})$ | $O(1)$ |
| | <hr/> |
| | $O(1)$ |
| IENESPERA (in $s : \text{dcnet}$, in $c : \text{compu}$) $\longrightarrow res : \text{puntero}(\text{conjLogP}(\text{paquete}))$ | |
| $t : < \text{nat}, \text{conjLog}(\text{paquete}, <_{id}), \text{conjLog}(\text{paquete}, <_p),$ | |
| $\text{conjLog}(\text{paquetePos}, <_{id}), \text{conjLog}(\text{paquetePos}, <_p) >$ | |
| $t \leftarrow \text{OBTENER}(\pi_1(c), s.\text{CompusPorPref})$ | $O(L)$ |
| $res \leftarrow \&(\pi_3(t))$ | $O(1)$ |
| | <hr/> |
| | $O(L)$ |
| IAVANZARSEGUNDO (in/out $s : \text{dcnet}$) | |
| var $i : \text{nat}$ | |
| $i \leftarrow 0$ | $O(1)$ |
| var $m : \text{nat}$ | |
| $m \leftarrow s.\text{Compus}[\text{LaQMasEnvio}].\#PaqE$ | |
| while $i < \text{LONGITUD}(s.\text{Compus})$ do | $O(n)$ |
| var $IP : \text{String}$ | |
| $IP \leftarrow s.\text{Compus}[i].IP$ | |
| $t_1 : < \text{nat}, \text{conjLog}(\text{paquete}, <_{id}), \text{conjLog}(\text{paquete}, <_p),$ | |
| $\text{conjLog}(\text{paquetePos}, <_{id}), \text{conjLog}(\text{paquetePos}, <_p) >$ | |
| $t_1 \leftarrow \text{OBTENER}(IP, s.\text{CompusPorPref})$ | $O(L)$ |
| var $p : \text{paquete}$ | |
| if $\neg \text{VACIA}?(\pi_5(t_1))$ then | |
| $p' \leftarrow \text{SACARMAX}(\pi_5(t_1))$ | $O(\log(k))$ |
| $\text{BORRAR}(\pi_2(t_1), \text{PAQUETE}(p'))$ | $O(\log(k))$ |
| $\text{BORRAR}(\pi_3(t_1), \text{PAQUETE}(p'))$ | $O(\log(k))$ |
| $\text{BORRAR}(\pi_4(t_1), p')$ | $O(\log(k))$ |
| $\text{BORRAR}(\pi_5(t_1), p')$ | $O(\log(k))$ |
| $s.\text{Compus}[i].\#PaqE \leftarrow s.\text{compus}[i].\#PaqE + 1$ | $O(1)$ |
| $\text{proxima} \leftarrow s.\text{CaminosMinimos}[\text{INDICEORIGEN}(p')][\text{INDICEDESTINO}(p')][\text{INDICEACTUAL}(p') +$ | |
| $1]$ | $O(1)$ o $O(L)$ si se copia |
| ACLARACION: coherente con caminoMinimo (sup que da un arreglo de IP) | |
| if $\neg (\text{DESTINO}(\text{PAQUETE}(p')) = \text{proxima})$ then | $O(L)$ |
| ACLARACION: Si no es porque ya no esta en transito entonces hay que aclarar en algún lugar si eliminamos el paquete | |

| | |
|--|-----------------------------|
| ACTUALIZARINDICE(p') | $O(1)$ |
| $t_2 \leftarrow \text{nat}, \text{conjLog}(\text{paquete}, <_{id}), \text{conjLog}(\text{paquete}, <_p),$ $\text{conjLog}(\text{paquetePos}, <_{id}), \text{conjLog}(\text{paquetePos}, <_p) >$ | |
| $t_2 \leftarrow \text{OBTENER}(\text{proxima}, s.\text{CompusPorPref})$ | $O(L)$ |
| INSERTAR($\pi_2(t_2), \text{PAQUETE}(p')$) | $O(\log(k))$ |
| INSERTAR($\pi_3(t_2), \text{PAQUETE}(p')$) | $O(\log(k))$ |
| INSERTAR($\pi_4(t_2), p'$) | $O(\log(k))$ |
| INSERTAR($\pi_5(t_2), p'$) | $O(\log(k))$ |
| end if | |
| if $s.\text{Compus}[i].\#PaqE > \text{max}$ then | $O(1)$ |
| $\text{max} \leftarrow i$ | $O(1)$ |
| end if | |
| end if | |
| $i \leftarrow i + 1$ | $O(1)$ |
| end while | |
| $s.\text{LaQMasEnvio} \leftarrow \text{max}$ | $O(1)$ |
| <hr/> | |
| | $O(n \times (L + \log(k)))$ |

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

| | |
|--|--------|
| var $i : \text{nat}$ | |
| $i \leftarrow 0$ | $O(1)$ |
| while $s.\text{compus}[i].IP \neq \pi_1(c)$ do | $O(n)$ |
| $i \leftarrow i + 1$ | $O(1)$ |
| end while | |
| $res \leftarrow s.\text{compus}[i].\#PaqE$ | $O(1)$ |
| <hr/> | |
| | $O(n)$ |

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

| | |
|--|-----------------------|
| var $i : \text{nat}$ | |
| $i \leftarrow 0$ | $O(1)$ |
| var $b : \text{bool}$ | |
| $b \leftarrow \neg(\text{PERTENECE?}(p, *(s.\text{compus}[i].pN)))$ ESTA BIEN ESTE PERTENECE? Y CREO que la estructura deberia apuntar a colalog con paquetespos tambien | $O(\log(k))$ |
| while b do | $O(n)$ |
| $i \leftarrow i + 1$ | $O(1)$ |
| $b \leftarrow \neg(\text{PERTENECE?}(p, *(s.\text{compus}[i].pN)))$ | $O(\log(k))$ |
| end while | |
| $res \leftarrow s.\text{CaminosMinimos}[\text{INDICEORIGEN}(p')][i]$ | |
| <hr/> | |
| | $O(n \times \log(k))$ |

1.3 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 Módulo 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() \rightarrow res : red

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

Post $\equiv \{\text{res} \equiv \text{iniciarRed}\}$

Descripción: Crea una red vacía

Complejidad: O(?)

Aliasing:

AGREGARCOMPU(**in/out** r : red, **in** c : compu)

Pre $\equiv \{r \equiv r_0 \wedge (\forall c_1:\text{compu})(c_1 \in \text{computadoras}(r)) \Rightarrow \text{IP}(c) \neq \text{IP}(c_1)\}$

Post $\equiv \{r \equiv \text{agregarComputadora}(r, c)\}$

Descripción: Agrega a la red r la computadora c

Complejidad: O(?)

Aliasing:

CONECTAR(**in/out** r : red, **in** c1 : compu, **in** i1 : interfaz, **in** c2 : compu, **in** i2 : interfaz)

Pre $\equiv \{r \equiv r_0 \wedge c_1 \in \text{computadoras}(r) \wedge c_2 \in \text{computadoras}(r) \wedge \text{IP}(c1) \neq \text{IP}(c2) \\ \wedge \neg \text{conectadas}(r, c1, c2) \wedge \neg \text{UsaInterfaz?}(r, c1, i1) \wedge \neg \text{UsaInterfaz?}(r, c2, i2)\}$

Post $\equiv \{r \equiv \text{conectar}(r, c1, i1, c2, i2)\}$

Descripción: Conecta las computadoras c1 y c2

Complejidad: O(?)

Aliasing:

CAMINOSMINIMOS(**in** r : red, **in** c1 : compu, **in** c2 : compu) \rightarrow res : conj(secu(nat))

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

Post $\equiv \{r \equiv \text{caminosMinimos}(r, c1, c2)\}$

Descripción: Devuelve los caminos mínimos entre dos computadoras

Complejidad: O(?)

Aliasing:

STRINGAINDICE(**in** r : red, **in** c : compu) \rightarrow res : nat

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

Post $\equiv \{\text{res} \equiv \text{indice}(r, c)\}$

Descripción: Devuelve un nat distinto para cada IP

Complejidad: O(?)

Aliasing:

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

Pre $\equiv \{c \in \text{computadoras}(r) \wedge i \in \text{interfaces}(c)\}$

Post $\equiv \{\text{res} \equiv \text{UsaInterfaz?}(r, c, i)\}$

Descripción: Devuelve TRUE si y solo si la interfaz está siendo utilizada

Aliasing:

$$\mathbf{Pre} \equiv \{c_1 \in \text{computadoras}(r) \wedge c_2 \in \text{computadoras}(r)\}$$

Descripción: Devuelve TRUE sí y solo sí hay un camino válido entre las dos computadoras

Complejidad: $O(?)$

Aliasing:

$$\text{indice} : \text{red} \times \text{copmu} \longrightarrow \text{nat}$$
$$\text{indice}(\text{iniciarRed}, c) \equiv 0$$
$$\text{indice}(\text{conectar}(r, c_1, i_1, c_2, i_2), c_3) \equiv 0$$
$$\text{indice}(\text{agregarComputadora}(r, c_1), c_2) \equiv$$

if $c_1 = c_2$ then

$$1 + \text{indice}(r, c_2)$$

else

$$\text{indice}(r, c_2)$$

Red está representada por un conjunto de computadoras y un puntero al primer Nodo, que coincide con la primer computadora que se agrega. El Nodo está representado por el IP de la computadora, un puntero a la próxima computadora, que forma una lista circular que conecta todas las computadoras agregadas y un arreglo de punteros que simboliza las interfaces y a dónde se conectan. Para el modulo red, se podría haber agregado un modulo intermedio de abstracción (por ejemplo con un diccionario), pero no aportaba mucho al diseño modular. Tampoco facilitaba mucho la escritura del Invariante. Por eso, y para conciliar tiempo disponible con calidad de diseño, decidimos representarlo como se muestra a continuación.

se representa con `redstr`

donde `redstr` es tupla \langle Compus : conj(compu),
Web : puntero(Nodo) \rangle

donde `Nodo` es `tupla(IP : String,`
`Proxima : puntero(Nodo),`
`Conexiones : arreglo_dimensionable de puntero(Nodo))`

donde `compu` es `tupla⟨IP : String,`
`Interfaces : conj(nat)⟩`

1. El conjunto Compus es vacío sí y solo sí Web apunta a NULL.
2. Partiendo del Nodo apuntado por Web, y avanzando por Proxima, se vuelve al Nodo apuntado por Web.
3. Partiendo del Nodo apuntado por Web, y avanzando por Proxima, cada nodo corresponde a una computadora:
El proyector IP de Nodo de todos los nodos conectados avanzando por Proxima, partiendo

desde el Nodo apuntado por Web pertenece al conjunto formado por la unión de todos los proyectores IP de compu, para todas las compu en Compus.

4. El tamaño del arreglo corresponde a la cantidad de interfaces:

El proyector Conexiones de Nodo de todos los nodos conectados avanzando por Proxima, partiendo desde el Nodo apuntado por Web es un arreglo cuya longitud es igual al cardinal del conjunto Interfaces de la compu cuyo proyector IP es igual al proyector IP del Nodo formado por dicho proyector Conexiones, para todas las compu en Compus.

5. Si la compu a está conectada con b , entonces b está conectada con a :

Para todos los punteros del proyector Conexiones de Nodo de todos los nodos conectados avanzando por Proxima, partiendo desde el Nodo apuntado por Web, el Nodo al que apunta dicho puntero tiene en alguna posición de su proyector Conexiones un puntero al Nodo en el cual esta el arreglo del cual el primer puntero forma parte.

Función de abstracción

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

$$(\forall e : \widehat{\text{redstr}})$$

$$\text{Abs}(e) \equiv r : \widehat{\text{red}} \mid$$

$$\text{computadoras}(r) =_{\text{obs}} e.\text{Compus} \wedge$$

$$\text{conectadas?}(r, c1, c2) =_{\text{obs}} \text{ `` Existe Nodo avanzando por Próximo, tal que `` } (\exists i:\text{nat})(0 \leq i \leq$$

$$\text{tam}(\text{Conexiones}) \wedge \text{IP}(\text{Nodo}) = \text{IP}(c1)) \Rightarrow \text{IP}(*\text{Conexiones}[i]) = \text{IP}(c2) \wedge$$

$$\text{interfazUsada}(r, c1, c2) =_{\text{obs}} i \text{ --- `` Existe un par de nodos avanzando por Próximo, tal que `` }$$

$$(\text{IP}(\text{Nodo1}) = \text{IP}(c1) \wedge \text{IP}(\text{Nodo2}) = \text{IP}(c2)) \Rightarrow \text{Conexiones}(\text{Nodo1})[i] = \text{Nodo2}$$

Algoritmos

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 $CONJ(\alpha)$

géneros $conjLog(\alpha)$

Operaciones

NUEVO(\bullet) $\rightarrow res : conjLog(\alpha)$

Pre $\equiv \{true\}$

Post $\equiv \{res =_{obs} \emptyset\}$

Descripción: Crea un nuevo conjLog vacío

Complejidad: $O(1)$

VACÍO?(**in** $cl : conjLog(\alpha)$) $\rightarrow res : bool$

Pre $\equiv \{true\}$

Post $\equiv \{res = (\emptyset?(cl))\}$

Descripción: Indica si el conjunto es vacío

Complejidad: $O(\log(\#(cl)))$

PERTENECE(**in** $cl : conjLog(\alpha)$, **in** $e : \alpha$) $\rightarrow res : bool$

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** $cl : conjLog(\alpha)$, **in** $e : \alpha$) $\rightarrow res : \alpha$

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** $cl : conjLog(\alpha)$, **in** $e : \alpha$) $\rightarrow res : \alpha$

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** $cl : cl(\alpha)$, **in** $e : \alpha$)

Pre $\equiv \{cl_0 =_{obs} cl \wedge \neg(e \in cl)\}$

Post $\equiv \{cl_0 =_{obs} Agregar(cl_0, e)\}$

Descripción: Inserta un nuevo elemento en el conjunto

Complejidad: $O(\log(\#(cl)))$

BORRAR(**in/out** $cl : cl(\alpha)$, *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 árbol 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

$$\begin{aligned} \text{Abs} : \widehat{\text{clog}(\alpha)} \text{ } cl &\longrightarrow \widehat{\text{conj}(\alpha)} && \{\text{Rep}(cl)\} \\ (\forall cl : \widehat{\text{clog}(\alpha)}) & \\ \text{Abs}(cl) \equiv c : \widehat{\text{conj}(\alpha)} & \mid \\ ((\forall e : \alpha) e \in c \Rightarrow_L \text{esta}(cl, e)) \wedge \text{size}(cl) = \#(c) & \end{aligned}$$

3.4.1 Aclaración de complejidades

Este módulo es un AVL y fue diseñado para poder cumplir con las especificaciones de complejidad de dcnets. 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, asignación y copia del tipo paramétrico α deben poder ser realizadas en $O(1)$.

3.5 Algoritmos

| | |
|---|----------------------------|
| IVACÍO? (in $cl : \text{conjLog}(\alpha)$) $\longrightarrow res : \text{bool}$ | |
| $res \leftarrow cl == \text{NULL}$ | $O(1)$ |
| <hr/> | |
| IBORRAR (in/out $cl : \text{conjLog}(\alpha)$, <i>in</i> $e : \alpha$) | $O(1)$ |
| var <i>variandoHijoDerecho?</i> : $\text{bool} \leftarrow \text{true}$ | |
| var <i>clactual</i> : $\text{conjLog}(\alpha) \leftarrow cl$ | $O(1)$ |
| var <i>aBorrar</i> : $\text{conjLog}(\alpha)$ | |
| if ($\neg(cl.der == \text{NULL}) \wedge \neg(cl.izq == \text{NULL})$) then | $O(1)$ |
| <i>clactual</i> $\leftarrow \text{IENCONTRARPADRE}(clactual, e)$ | $O(\log(\text{size}(cl)))$ |
| if <i>clactual.der</i> ! = $\text{NULL} \wedge_L clactual.der.valor == e$ then | |
| <i>aBorrar</i> $\leftarrow clactual.der$ | $O(1)$ |
| else | |
| <i>aBorrar</i> $\leftarrow clactual.izq$ | $O(1)$ |
| end if | |
| var <i>mm</i> : $\text{conjLog}(\alpha) \leftarrow \text{IDAMEMAYORMENORES}(clactual)$ | $O(\log(\text{size}(cl)))$ |
| if <i>mm.valor</i> == e then | $O(1)$ |
| if <i>mm.padre.der</i> ! = $\text{NULL} \wedge_L mm.padre.der.valor == mm.valor$ then | |
| <i>variandoHijoDerecho?</i> $\leftarrow \text{true}$ | $O(1)$ |
| <i>mm.padre.der</i> = NULL | |
| <i>mm.padre.fdb</i> – – | |
| else | |
| <i>variandoHijoDerecho?</i> $\leftarrow \text{false}$ | $O(1)$ |
| <i>mm.padre.izq</i> = NULL | |
| <i>mm.padre.fdb</i> ++ | |
| end if | |
| else | |
| var <i>mmValor</i> : $\alpha \leftarrow mm.valor$ | $O(1)$ |
| <i>aBorrar.valor</i> $\leftarrow mmValor$ | $O(1)$ |
| if <i>mm.izq</i> ! = NULL then | $O(1)$ |
| <i>mm.valor</i> $\leftarrow mm.izq.valor$ | $O(1)$ |
| <i>mm.izq</i> $\leftarrow \text{NULL}$ | $O(1)$ |
| <i>mm.fdb</i> ++ | $O(1)$ |
| if <i>mm.padre.valor</i> == e then | |
| <i>variandoHijoDerecho?</i> $\leftarrow \text{false}$ | |
| else | |
| <i>variandoHijoDerecho?</i> $\leftarrow \text{true}$ | |
| end if | |
| else | |

```

    if  $mm.padre.valor == e$  then
         $mm.padre.izq = NULL$ 
         $variandoHijoDerecho? \leftarrow false$ 
    else
         $mm.padre.der = NULL$ 
         $variandoHijoDerecho? \leftarrow true$ 
    end if
end if
end if
iREBYRECALCFDB( $mm.padre, variandoHijoDerecho?, estoyBorrando?$ )
                                          $O(\log(size(cl)))$ 
else

    if  $cl.der == NULL \wedge cl.izq == NULL$  then
         $cl \leftarrow NULL$ 
    else

        if  $cl.der == NULL$  then

            if  $cl.izq.valor == e$  then
                 $cl.izq \leftarrow NULL$ 
            else
                 $cl.valor \leftarrow cl.izq.valor$ 
                 $cl.izq \leftarrow NULL$ 
            end if
        else

            if  $cl.der.valor == e$  then
                 $cl.der \leftarrow NULL$ 
            else
                 $cl.valor \leftarrow cl.der.valor$ 
                 $cl.der \leftarrow NULL$ 
            end if
        end if
    end if
end if
end if

```

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

Justificación de complejidad

Por álgebra de órdenes

iINSERTAR(in/out $cl : conjLog(\alpha)$, in $e : \alpha$)

```

var  $clactual : conjLog(\alpha) \leftarrow cl$ 
                                          $O(1)$ 

if  $\neg(cl.der == NULL) \wedge \neg(cl.izq == NULL)$  then
     $clactual \leftarrow iENCONTRARPADRE(clactual, e)$ 
                                          $O(\log(size(cl)))$ 

    if  $clactual.valor < e$  then

```

| | |
|--|---|
| <pre> <i>clactual.der</i> ← tupla(<i>der</i> : <i>NULL</i>, <i>izq</i> : <i>NULL</i>, <i>valor</i> : <i>e</i>, <i>padre</i> : <i>clactual</i>, <i>fdb</i> : 0) iREBYRECALCFDB(<i>clactual</i>, <i>true</i>, <i>false</i>) else <i>clactual.izq</i> ← tupla(<i>der</i> : <i>NULL</i>, <i>izq</i> : <i>NULL</i>, <i>valor</i> : <i>e</i>, <i>padre</i> : <i>clactual</i>, <i>fdb</i> : 0) iREBYRECALCFDB(<i>clactual</i>, <i>false</i>, <i>false</i>) end if else if <i>cl.der</i> == <i>NULL</i> ∧ <i>cl.izq</i> == <i>NULL</i> then <i>cl</i> ← tupla(<i>der</i> : <i>NULL</i>, <i>izq</i> : <i>NULL</i>, <i>valor</i> : <i>e</i>, <i>padre</i> : <i>clactual</i>, <i>fdb</i> : 0) else if <i>cl.der</i>! = <i>NULL</i> then <i>cl.izq</i> ← tupla(<i>der</i> : <i>NULL</i>, <i>izq</i> : <i>NULL</i>, <i>valor</i> : <i>e</i>, <i>padre</i> : <i>cl</i>, <i>fdb</i> : 0) else <i>cl.der</i> ← tupla(<i>der</i> : <i>NULL</i>, <i>izq</i> : <i>NULL</i>, <i>valor</i> : <i>e</i>, <i>padre</i> : <i>cl</i>, <i>fdb</i> : 0) end if end if end if </pre> | <div style="display: flex; flex-direction: column; align-items: flex-end;"> <div>$O(1)$</div> <div>$O(1)$</div> <div>$O(1)$</div> <div>$O(1)$</div> <div>$O(1)$</div> <div>$O(1)$</div> <div>$O(1)$</div> <div>$O(1)$</div> <div>$O(1)$</div> </div> |
| | <hr style="border: 0.5px solid black; margin-bottom: 5px;"/> $O(\log(\text{size}(cl)))$ |

Justificación de complejidad

Por álgebra de órdenes

$\text{iPERTENECE}(\text{in/out } cl : \text{conjLog}(\alpha), \text{ in } e : \alpha) \longrightarrow res : \text{bool}$

| | |
|--|--------|
| var <i>encontrado?</i> : <i>bool</i> ← <i>false</i> | $O(1)$ |
| var <i>clactual</i> : <i>conjLog</i> (α) ← <i>cl</i> | $O(1)$ |
| while (<i>clactual</i> ! = <i>NULL</i>) ∧ ¬(<i>encontrado?</i>) 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 | |
| $clactual \leftarrow NULL$ | $O(1)$ |
| $res \leftarrow encontrado?$ | $O(1)$ |
| <hr/> | |
| $O(\log(size(cl)))$ | |

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 $n = size(cl)$

$IMENOR(in\ cl : conjLog(\alpha)) \longrightarrow res : \alpha$

| | |
|---|---------------------|
| var $clactual : conjLog(\alpha) \leftarrow cl$ | $O(1)$ |
| $clactual \leftarrow iMenorNodo(clactual)$ | $O(\log(size(cl)))$ |
| $res \leftarrow clactual.valor$ | $O(1)$ |
| <hr/> | |
| $O(\log(size(cl)))$ | |

Justificación de complejidad

Por álgebra de órdenes

$IBUSCAR(in\ cl : conjLog(\alpha),\ e : \alpha) \longrightarrow res : \alpha$

| | |
|---|---------------------|
| var $padre : conjLog(\alpha) \leftarrow iEncontrarPadre(cl, e)$ | $O(\log(size(cl)))$ |
| if $padre.der \neq NULL \wedge_L padre.der.valor == e$ then | $O(1)$ |
| $res \leftarrow padre.der.valor$ | |
| else | |
| $res \leftarrow padre.izq.valor$ | |
| end if | |
| <hr/> | |
| $O(\log(size(cl)))$ | |

Justificación de complejidad

Por álgebra de órdenes

3.6 Auxiliares

IREBYRECALCFDB(**in/out** $cl : \text{conjLog}(\alpha)$, *in variandoHijoDerecho?* : **bool**, *in estoyBorrando?* : **bool**)

var $clactual : \text{conjLog}(\alpha) \leftarrow cl$ $O(1)$

var $termino? : \text{bool} \leftarrow false$ $O(1)$

if *estoyBorrando?* **then**

while $clactual! = NULL \wedge \neg(termino?)$ **do** $O(1)$

if *variandoHijoDerecho?* **then**

if $clactual.fdb == -1$ **then** $O(1)$

var $fdbIzq : \text{nat}$ $O(1)$

if $cl.izq! = NULL$ **then**
 $fdbIzq \leftarrow cl.izq$ $O(1)$
end if

if $cl.izq! = NULL \wedge_L cl.izq.fdb == 1$ **then** $O(1)$
iROTARLR($cl.izq$) $O(1)$
end if
iROTARLL(cl) $O(1)$

if $cl.izq! = NULL \wedge_L fdbIzq == 0$ **then** $O(1)$
 $termino? \leftarrow true$ $O(1)$
end if

else

if $cl.fdb == +1$ **then** $O(1)$
 $cl.fdb \leftarrow 0$ $O(1)$
 $termino? \leftarrow true$ $O(1)$

else
 $cl.fdb \leftarrow -1$ $O(1)$

end if

end if

else

if $clactual.fdb == -1$ **then** $O(1)$

var $fdbDer : \text{nat}$ $O(1)$

if $cl.der! = NULL$ **then** $O(1)$
 $fdbDer \leftarrow cl.der.fdb$ $O(1)$
end if

if $cl.der! = NULL \wedge_L fdbDer == 1$ **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 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)
            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
    end if
end while

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

```

| | |
|---|---------------------|
| if $cl.izq! = NULL$ then | $O(1)$ |
| $fdbIzq \leftarrow cl.izq.fdb$ | |
| end if | |
| if $cl.izq! = NULL \wedge_L fdbIzq == +1$ then | $O(1)$ |
| $iROTARLR(cl.izq)$ | $O(1)$ |
| end if | |
| $iROTARLL(cl)$ | $O(1)$ |
| $termino? \leftarrow true$ | $O(1)$ |
| else | |
| if $clactual.fdb == +1$ then | $O(1)$ |
| $clactual.fdb \leftarrow 0$ | $O(1)$ |
| $termino? \leftarrow true$ | $O(1)$ |
| else | |
| $clactual.fdb \leftarrow -1$ | $O(1)$ |
| end if | |
| end if | |
| end if | |
| $variandoHijoDerecho \leftarrow (cl.padre! = NULL \wedge_L cl.padre.der.valor == cl.valor)$ | $O(1)$ |
| $clactual \leftarrow clactual.padre$ | $O(1)$ |
| 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

$iROTARRR(\text{in/out } cl : \text{conjLog}(\alpha))$

| | |
|---|--------|
| var $nietoRR : \text{conjLog}(\alpha) \leftarrow cl.der.der$ | $O(1)$ |
| var $hijoDer : \text{conjLog}(\alpha) \leftarrow cl.der$ | $O(1)$ |
| var $hijoIzq : \text{conjLog}(\alpha) \leftarrow cl.izq$ | $O(1)$ |
| $cl.der \leftarrow NULL$ | $O(1)$ |
| $cl.izq = \text{tupla}(\text{der} : \text{hijoDer.der},$ | $O(1)$ |
| $izq : cl.izq,$ | |
| $valor : cl.valor,$ | |
| $padre : cl,$ | |
| $fdb : 0)$ | |
| $cl.izq.izq.padre \leftarrow cl.izq$ | $O(1)$ |
| $cl.izq.der.padre \leftarrow cl.izq$ | $O(1)$ |
| $cl.valor = hijoDer.valor$ | $O(1)$ |
| $cl.der = nietoRR$ | $O(1)$ |
| $cl.der.padre \leftarrow cl$ | $O(1)$ |
| | $O(1)$ |

Justificación de complejidad

Son operaciones sobre α y punteros

IROTARRL(**in/out** $cl : \text{conjLog}(\alpha)$)

| | |
|--|--------|
| var <i>nietoRR</i> : $\text{conjLog}(\alpha) \leftarrow cl.der.der$ | $O(1)$ |
| var <i>nietoRL</i> : $\text{conjLog}(\alpha) \leftarrow cl.der.izq$ | $O(1)$ |
| var <i>valorDer</i> : $\alpha \leftarrow cl.der.valor$ | $O(1)$ |
| <i>cl.der.valor</i> \leftarrow <i>nietoRL.valor</i> | $O(1)$ |
| <i>nietoRR.izq</i> \leftarrow <i>nietoRL.der</i> | $O(1)$ |
| <i>cl.der.der</i> \leftarrow <i>nietoRR</i> | $O(1)$ |
| <i>cl.der.izq</i> \leftarrow <i>nietoRL.izq</i> | $O(1)$ |
| <i>cl.der.der.padre</i> \leftarrow <i>cl.der</i> | $O(1)$ |
| <i>cl.der.izq.padre</i> \leftarrow <i>cl.der</i> | $O(1)$ |
| <i>cl.izq.fdb</i> $\leftarrow +1$ | $O(1)$ |
| <hr/> | |
| | $O(1)$ |

Justificación de complejidad

Son operaciones sobre α y punteros

IROTARLL(**in/out** $cl : \text{conjLog}(\alpha)$)

| | |
|---|--------|
| var <i>nietoLL</i> : $\text{conjLog}(\alpha) \leftarrow cl.izq.izq$ | $O(1)$ |
| var <i>hijoIzq</i> : $\text{conjLog}(\alpha) \leftarrow cl.izq$ | $O(1)$ |
| var <i>hijoDer</i> : $\text{conjLog}(\alpha) \leftarrow cl.der$ | $O(1)$ |
| <i>cl.izq</i> \leftarrow <i>NULL</i> | $O(1)$ |
| <i>cl.der</i> = <i>tupla</i> (<i>der</i> : <i>cl.der</i> , <i>izq</i> : <i>hijoIzq.der</i> , <i>valor</i> : <i>cl.valor</i> , <i>padre</i> : <i>cl</i> , <i>fdb</i> : 0) | $O(1)$ |
| <i>cl.der.izq.padre</i> \leftarrow <i>cl.der</i> | $O(1)$ |
| <i>cl.der.der.padre</i> \leftarrow <i>cl.der</i> | $O(1)$ |
| <i>cl.valor</i> = <i>hijoIzq.valor</i> | $O(1)$ |
| <i>cl.izq</i> = <i>nietoLL</i> | $O(1)$ |
| <i>cl.izq.padre</i> \leftarrow <i>cl</i> | $O(1)$ |
| <hr/> | |
| | $O(1)$ |

Justificación de complejidad

Son operaciones sobre α y punteros

IROTARLR(**in/out** $cl : \text{conjLog}(\alpha)$)

| | |
|--|--------|
| var <i>nietoLL</i> : $\text{conjLog}(\alpha) \leftarrow cl.izq.izq$ | $O(1)$ |
|--|--------|

| | |
|---|--------|
| var <i>nietoLR</i> : $\text{conjLog}(\alpha) \leftarrow \text{cl.izq.der}$ | $O(1)$ |
| var <i>valorIzq</i> : $\alpha \leftarrow \text{cl.izq.valor}$ | $O(1)$ |
| <i>cl.izq.valor</i> $\leftarrow \text{nietoRL.valor}$ | $O(1)$ |
| <i>nietoLL.izq</i> $\leftarrow \text{nietoLR.izq}$ | $O(1)$ |
| <i>cl.izq.izq</i> $\leftarrow \text{nietoLL}$ | $O(1)$ |
| <i>cl.izq.der</i> $\leftarrow \text{nietoLR.der}$ | $O(1)$ |
| <i>cl.izq.izq.padre</i> $\leftarrow \text{cl.izq}$ | $O(1)$ |
| <i>cl.izq.der.padre</i> $\leftarrow \text{cl.izq}$ | $O(1)$ |
| <i>cl.izq.fdb</i> $\leftarrow -1$ | $O(1)$ |
| <hr/> | |
| | $O(1)$ |

Justificación de complejidad

Son operaciones sobre α y punteros

$\text{IENCONTRARPADRE}(\text{in } cl : \text{conjLog}(\alpha), e : \alpha) \longrightarrow res : \text{conjLog}(\alpha)$

| | |
|---|----------------------|
| var <i>clactual</i> : $\text{conjLog}(\alpha)$ | $O(1)$ |
| var <i>encontrado?</i> : $\text{bool} \leftarrow (\text{clactual.der!} = \text{NULL} \wedge_L \text{clactual.der.valor} == e) \vee (\text{clactual.izq!} = \text{NULL} \wedge_L \text{clactual.izq.valor} == e)$ | |
| while $\neg \text{encontrado?}$ do | |
| if $e > \text{clactual.valor}$ then | |
| <i>clactual</i> $\leftarrow \text{clactual.der}$ | $O(1)$ |
| else | |
| <i>clactual</i> $\leftarrow \text{clactual.izq}$ | $O(1)$ |
| end if | |
| <i>encontrado?</i> $\leftarrow (\text{clactual.der!} = \text{NULL} \wedge_L \text{clactual.der.valor} == e) \vee (\text{clactual.izq!} = \text{NULL} \wedge_L \text{clactual.izq.valor} == e)$ | |
| end while | |
| <i>res</i> $\leftarrow \text{clactual}$ | $O(1)$ |
| <hr/> | |
| | $O(\text{size}(cl))$ |

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 $n = \text{size}(cl)$

$\text{IDAMEMAYORMENORES}(\text{in } cl : \text{conjLog}(\alpha), e : \alpha) \longrightarrow res : \text{conjLog}(\alpha)$

| | |
|---|----------------------------|
| var <i>clactual</i> : $\text{conjLog}(\alpha) \leftarrow cl$ | $O(1)$ |
| if $\text{clactual.izq!} = \text{NULL}$ then | $O(1)$ |
| <i>clactual</i> $\leftarrow \text{iMayorNodo}(\text{clactual})$ | $O(\log(\text{size}(cl)))$ |
| end if | |
| <i>res</i> $\leftarrow \text{clactual}$ | $O(1)$ |
| <hr/> | |

Justificación de complejidad

Por álgebra de órdenes

$\text{IMENORNODO}(\text{in } cl : \text{conjLog}(\alpha)) \longrightarrow res : \text{conjLog}(\alpha)$

var $clactual : \text{conjLog}(\alpha) \leftarrow cl$ $O(1)$

while $clactual.izq! = NULL$ **do**

$clactual \leftarrow clactual.izq$

end while

$res \leftarrow clactual$

$O(1)$

$O(\log(\text{size}(cl)))$

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

$\text{IMAYORNODO}(\text{in } cl : \text{conjLog}(\alpha)) \longrightarrow res : \text{conjLog}(\alpha)$

var $clactual : \text{conjLog}(\alpha) \leftarrow cl$ $O(1)$

while $clactual.der! = NULL$ **do**

$clactual \leftarrow clactual.der$

end while

$res \leftarrow clactual$

$O(1)$

$O(\log(\text{size}(cl)))$

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

$\text{SIZE}(\text{in } cl : \text{conjLog}(\alpha)) \longrightarrow res : nat$

if $cl == NULL$ **then**

$res \leftarrow 0$

else

$res \leftarrow 1 + iSize(cl.der) + iSize(cl.izq)$

end if

3.7 Operaciones auxiliares de $\text{conj}(\alpha)$

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

$menor(c) =$

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 $\text{DICCIONARIO}(\text{SECU}(\text{CHAR}), \beta)$

géneros $\text{diccPref}(\text{secu}(\text{char}), \beta)$

Operaciones

$\text{NUEVO}() \longrightarrow \text{res} : \text{diccPref}(\text{secu}(\text{char}), \beta)$

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

Post $\equiv \{(\forall p : \text{secu}(\text{char})) \neg(\text{def?}(p, \text{res}))\}$

Descripción: Crea un nuevo diccionario vacío

Complejidad: $O(1)$

$\text{DEF?}(\text{in } dp : \text{diccPref}(\text{secu}(\text{char}), \beta), \text{ in } p : \text{secu}(\text{char})) \longrightarrow \text{res} : \text{bool}$

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

Post $\equiv \{\text{res} = p \in \text{claves}(dp)\}$

Descripción: Devuelve true o false según si la clave está o no definida

Complejidad: $O(L)$

$\text{CLAVES}(\text{in } dp : \text{diccPref}(\text{secu}(\text{char}), \beta)) \longrightarrow \text{res} : \text{conj}(\text{secu}(\text{char}))$

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

Post $\equiv \{(\forall c : \text{secu}(\text{char})) c \in \text{claves}(dp) \iff \text{def?}(c, dp)\}$

Descripción: Devuelve un conjunto de las claves del diccionario

Complejidad: $O(L)?$

$\text{DEFINIR}(\text{in/out } dp : \text{diccPref}(\text{secu}(\text{char}), \beta), \text{ in } p : \text{secu}(\text{char}), \text{ in } s : \beta)$

Pre $\equiv \{dp = dp_0 \wedge \neg \text{def?}(p, dp)\}$

Post $\equiv \{\text{def?}(p, dp) \wedge \text{obtener}(p, dp) =_{\text{obs}} s \wedge (\forall c \in \text{claves}(dp_0)) \text{def?}(c, dp)\}$

Descripción: Inserta una nueva clave con su significado en el diccionario

Complejidad: $O(L)$

$\text{OBTENER}(\text{in } dp : \text{diccPref}(\text{secu}(\text{char}), \beta), \text{ in } p : \text{secu}(\text{char})) \longrightarrow \text{res} : \beta$

Pre $\equiv \{\text{def?}(p, dp)\}$

Post $\equiv \{\text{res} = \text{obtener}(p, dp)\}$

Descripción: Retorna el significado de la clave pedida

Complejidad: $O(L)$

Aliasing: Devuelve res por referencia

$\text{ELIMINAR}(\text{in/out } dp : \text{diccPref}(\text{secu}(\text{char}), \beta), \text{ in } p : \text{secu}(\text{char}))$

Pre $\equiv \{dp = dp_0 \wedge \text{def?}(p, dp)\}$

Post $\equiv \{\neg \text{def?}(p, dp) \wedge (\forall c \in \text{claves}(dp_0), c \neq p) \text{def?}(c, dp)\}$

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** $id : \text{nat}$, **in** $o : \text{compu}$, **in** $d : \text{compu}$, **in** $pr : \text{nat}$) $\longrightarrow res : \text{paquete}$

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

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: $O(1)$

• $<_p$ •(**in** $p_1 : \text{paquete}$, **in** $p_2 : \text{paquete}$) $\longrightarrow res : \text{bool}$

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

Post $\equiv \{res = \text{true} \iff (\pi_2(p_1) = \pi_2(p_2) \wedge pi_1(p_1) < pi_1(p_2) \vee (\pi_1(p_1) < \pi_1(p_2)))\}$

Descripción: Define un orden en paquete según la prioridad

Complejidad: $O(1)$

• $<_{id}$ •(**in** $p_1 : \text{paquete}$, **in** $p_2 : \text{paquete}$) $\longrightarrow res : \text{bool}$

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

Post $\equiv \{res = \text{true} \iff id(p_1) < id(p_2)\}$

Descripción: Define un orden en paquete según el id

Complejidad: $O(1)$

ID(**in** $p : \text{paquete}$) $\longrightarrow res : \text{nat}$

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

Post $\equiv \{res = \pi_1(paquete)\}$

Descripción: *Getterdeid*

Complejidad: $O(1)$

PRIORIDAD(**in** $p : \text{paquete}$) $\longrightarrow res : \text{nat}$

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

Post $\equiv \{res = \pi_2(paquete)\}$

Descripción: *Getterdeprioridad*

Complejidad: $O(1)$

ORIGEN(**in** $p : \text{paquete}$) $\longrightarrow res : \text{Ip}$

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

Post $\equiv \{res = \pi_3(paquete)\}$

Descripción: *Getterdeorigen*

Complejidad: $O(1)$

DESTINO(**in** $p : \text{paquete}$) $\longrightarrow res : \text{Ip}$

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

Post $\equiv \{res = \pi_4(paquete)\}$

Descripción: *Getterdedestino*

Complejidad: $O(1)$

5.2 Representación

se representa con paquete

donde paquete es tupla \langle id : nat,
 origen : Ip,
 destino : Ip,
 prioridad : nat \rangle

6 PaquetePos

Un PaquetePos es

6.1 Interfaz

se explica con $\text{tupla} \langle : \text{Paquete},$
 $\quad \quad \quad : \text{nat},$
 $\quad \quad \quad : \text{nat},$
 $\quad \quad \quad : \text{nat} \rangle$

géneros $\quad \quad \quad \text{paquetePos}$

Operaciones

$\text{CREARPAQUETE}(\text{in } id : \text{nat}, \text{in } o : \text{compu}, \text{in } d : \text{compu}, \text{in } pr : \text{nat}) \longrightarrow res : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle$

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

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: $O(1)$

$\bullet \langle_p \bullet (\text{in } p_1 : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle, \text{in } p_2 : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle) \longrightarrow res : \text{bool}$

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

Post $\equiv \{res = \text{true} \iff (\pi_2(\pi_1(p_1)) = \pi_2(\pi_1(p_2)) \wedge \pi_1(\pi_1(p_1)) < \pi_1(\pi_1(p_2)) \vee (\pi_1(\pi_1(p_1)) < \pi_1(\pi_1(p_2))) \vee (\pi_1(\pi_1(p_2)) < \pi_1(\pi_1(p_1))))\}$

Descripción: Define un orden en paquete según la prioridad

Complejidad: $O(1)$

$\bullet \langle_{id} \bullet (\text{in } p_1 : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle, \text{in } p_2 : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle) \longrightarrow res : \text{bool}$

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

Post $\equiv \{res = \text{true} \iff (pi_1(pi_1(p_1)) < (pi_1(pi_1(p_2))))\}$

Descripción: Define un orden en paquete según el id

Complejidad: $O(1)$

$\text{GETPAQUETE}(\text{in } ppos : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle) \longrightarrow res : \text{paquete}$

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

Post $\equiv \{res =_{\text{obs}} \pi_1(\text{paquete})\}$

Descripción: Getter de paquete

Complejidad: $O(1)$

$\text{INDICEORIGEN}(\text{in } p : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle) \longrightarrow res : \text{nat}$

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

Post $\equiv \{res = \pi_2(p)\}$

Descripción: Getter de indiceOrigen

Complejidad: $O(1)$

$\text{INDICEDESTINO}(\text{in } p : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle) \longrightarrow res : \text{nat}$

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

Post $\equiv \{res = \pi_3(p)\}$

Descripción: Getter de indiceDestino

Complejidad: $O(1)$

$\text{POSACTUAL}(\text{in } p : \langle \text{paquete}, \text{nat}, \text{nat}, \text{nat} \rangle) \longrightarrow res : \text{nat}$

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

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 posición 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.