

# Algoritmos y Estructuras de Datos II

Departamento de Computación  
Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

## Trabajo Práctico II

### Grupo: 12

| Integrante            | LU     | Correo electrónico            |
|-----------------------|--------|-------------------------------|
| Pondal, Iván          | 078/14 | ivan.pondal@gmail.com         |
| Paz, Maximiliano León | 251/14 | m4xileon@gmail.com            |
| Mena, Manuel          | 313/14 | manuelmena1993@gmail.com      |
| Demartino, Francisco  | 348/14 | demartino.francisco@gmail.com |

### Reservado para la cátedra

| Instancia       | Docente | Nota |
|-----------------|---------|------|
| Primera entrega |         |      |
| Segunda entrega |         |      |

# Índice

|   |           |
|---|-----------|
| <b>1. Módulo DCNet</b>  | <b>4</b>  |
| 1.1. Interfaz   | 4         |
| 1.1.1. Operaciones básicas de DCNet                               | 4         |
| 1.2. Representación   | 6         |
| 1.2.1. Representación de dcnet                                    | 6         |
| 1.2.2. Invariante de Representación                               | 6         |
| 1.2.3. Función de Abstracción                                     | 9         |
| 1.3. Algoritmos   | 9         |
| <b>2. Módulo Red</b>  | <b>14</b> |
| 2.1. Interfaz   | 14        |
| 2.2. Representación   | 15        |
| 2.2.1. Estructura   | 15        |
| 2.2.2. Invariante de Representación                               | 16        |
| 2.2.3. Función de Abstracción                                     | 18        |
| 2.3. Algoritmos   | 18        |
| <b>3. Módulo Cola de mínima prioridad(<math>\alpha</math>)</b>    | <b>24</b> |
| 3.1. Especificación   | 24        |
| 3.2. Interfaz   | 25        |
| 3.2.1. Operaciones básicas de Cola de mínima prioridad            | 25        |
| 3.3. Representación   | 26        |
| 3.3.1. Representación de colaMinPrior                             | 26        |
| 3.3.2. Invariante de Representación                               | 26        |
| 3.3.3. Función de Abstracción                                     | 26        |
| 3.4. Algoritmos   | 26        |
| <b>4. Módulo Diccionario logarítmico(<math>\alpha</math>)</b>     | <b>28</b> |
| 4.1. Interfaz   | 28        |
| 4.1.1. Operaciones básicas de Diccionario logarítmico( $\alpha$ ) | 28        |
| 4.1.2. Operaciones auxiliares del TAD                             | 29        |
| 4.2. Representación   | 29        |
| 4.2.1. Representación de diccLog( $\alpha$ )                      | 29        |
| 4.2.2. Invariante de Representación                               | 29        |
| 4.2.3. Función de Abstracción                                     | 30        |
| 4.3. Algoritmos   | 30        |
| <b>5. Módulo Árbol binario(<math>\alpha</math>)</b>               | <b>37</b> |
| 5.1. Interfaz   | 37        |
| 5.1.1. Operaciones básicas de Árbol binario( $\alpha$ )           | 37        |
| 5.2. Representación   | 38        |

---

|   |           |
|---|-----------|
| 5.2.1. Representación de $\text{ab}(\alpha)$ . . . . .          | 38        |
| 5.2.2. Invariante de Representación . . . . .                   | 38        |
| 5.2.3. Función de Abstracción . . . . .                         | 38        |
| 5.3. Algoritmos . . . . .                                       | 38        |
| <b>6. Módulo Diccionario <math>\text{String}(\alpha)</math></b> | <b>40</b> |
| 6.1. Interfaz . . . . .   | 40        |

# 1. Módulo DCNet

## 1.1. Interfaz

se explica con: DCNET.

géneros: dcnet.

### 1.1.1. Operaciones básicas de DCNet

INICIARDCNET(**in**  $r$ : red)  $\rightarrow res$  : dcnet

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

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

**Complejidad:**  $O(n * (n + L))$  donde n es la cantidad de computadoras y L es la longitud de nombre de computadora mas larga

**Descripción:** crea una DCNet nueva tomando una red

CREARPAQUETE(**in/out**  $dcn$ : dcnet, **in**  $p$ : paquete)

**Pre**  $\equiv \{$

$dcn =_{obs} dcn_0 \wedge$

$\neg( (\exists p':paquete)( paqueteEnTransito(dcn, p') \wedge id(p) = id(p') \wedge origen(p) \in computadoras(red(dcn)) \wedge_L$   
 $destino(p) \in computadoras(red(dcn)) \wedge_L hayCamino?(red(dcn), origen(p), destino(p))) )$

$\}$

**Post**  $\equiv \{dcn =_{obs} crearPaquete(dcn_0)\}$

**Complejidad:**  $O(L + \log(k))$  donde L es la longitud de nombre de computadora mas larga y k es la longitud de la cola de paquetes mas larga

**Descripción:** crea un nuevo paquete

AVANZARSEGUNDO(**in/out**  $dcn$ : dcnet)

**Pre**  $\equiv \{dcn =_{obs} dcn_0\}$

**Post**  $\equiv \{dcn =_{obs} avanzarSegundo(dcn_0)\}$

**Complejidad:**  $O(n * (L + \log(k)))$  donde n es la cantidad de computadoras, L es la longitud de nombre de computadora mas larga y k es la longitud de la cola de paquetes mas larga

**Descripción:** envia los paquetes con mayor prioridad a la siguiente compu

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

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

**Post**  $\equiv \{alias(res =_{obs} red(dcn))\}$

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

**Descripción:** devuelve la red de una DCNet

**Aliasing:** res es una referencia no modificable

CAMINORECORRIDO(**in**  $dcn$ : dcnet, **in**  $p$ : paquete)  $\rightarrow res$  : secu(compu)

**Pre**  $\equiv \{paqueteEnTransito?(dcn, p)\}$

**Post**  $\equiv \{alias(res =_{obs} caminoRecorrido(dcn, p))\}$

**Complejidad:**  $O(n * \log(k))$  donde n es la cantidad de computadoras y k es la longitud de la cola de paquetes mas larga

**Descripción:** devuelve el camino recorrido por un paquete

**Aliasing:** res es una referencia no modificable

CANTIDADENVIADOS(**in**  $dcn : \text{dcnet}$ , **in**  $c : \text{compu}$ )  $\rightarrow res : \text{nat}$

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

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

**Complejidad:**  $O(L)$  donde L es la longitud de nombre de computadora mas larga

**Descripción:** devuelve la cantidad de paquetes enviados por una compu

ENESPERA(**in**  $dcn : \text{dcnet}$ , **in**  $c : \text{compu}$ )  $\rightarrow res : \text{conj}(\text{paquete})$

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

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

**Complejidad:**  $O(L)$  donde L es la longitud de nombre de computadora mas larga

**Descripción:** devuelve el conjunto de paquetes encolados en una compu

**Aliasing:** res es una referencia no modificable

PAQUETEENTRANSITO(**in**  $dcn : \text{dcnet}$ , **in**  $p : \text{paquete}$ )  $\rightarrow res : \text{bool}$

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

**Post**  $\equiv \{res =_{\text{obs}} \text{paqueteEnTransito}(dcn, p)\}$

**Complejidad:**  $O(n * \log(k))$  donde n es es la cantidad de computadoras y k es la longitud de la cola de paquetes mas larga

**Descripción:** indica si el paquete está en transito

LAQUEMASENVIO(**in**  $dcn : \text{dcnet}$ )  $\rightarrow res : \text{compu}$

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

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{laQueMasEnvio}(dcn))\}$

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

**Descripción:** devuelve la compu que mas paquetes envió

**Aliasing:** res es una referencia no modificable

$\bullet = \bullet(\text{in } dcn_1 : \text{dcnet}, \text{in } dcn_2 : \text{dcnet}) \rightarrow res : \text{bool}$

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

**Post**  $\equiv \{res =_{\text{obs}} dcn_1 = dcn_2\}$

**Complejidad:**  $O(n * k^3 * (k + n))$

**Descripción:** compara  $dcn_1$  y  $dcn_2$  por igualdad

## 1.2. Representación

### 1.2.1. Representación de dcnet

dcnet se representa con estr

donde estr es tupla(*topología*: red,  
*vectorCompusDCNet*: vector(compuDCNet),  
*diccCompusDCNet*: diccString(puntero(compuDCNet)),  
*conjPaquetesDCNet*: conj(paqueteDCNet),  
*laQueMásEnvió*: puntero(compuDCNet))

donde compuDCNet es tupla(*pc*: puntero(compu),  
*conjPaquetes*: conj(paquete),  
*diccPaquetesDCNet*: diccLog(itConj(paqueteDCNet)),  
*colaPaquetesDCNet*: colaPrioridad(nat, itConj(paqueteDCNet)),  
*paqueteAEnviar*: itConj(paqueteDCNet), *enviados*: nat)

donde paqueteDCNet es tupla(*it*: itConj(paquete), *recorrido*: lista(compu))

donde paquete es tupla(*id*: nat, *prioridad*: nat, *origen*: compu, *destino*: compu)

donde compu es tupla(*ip*: string, *interfaces*: conj(nat))

### 1.2.2. Invariante de Representación

- (I) Las compus de los elementos de vectorCompusDCNet son punteros a todas las compus de la topología
- (II) Las claves de diccCompusDCNet son todos los hostnames de la topología
- (III) Los significados de diccCompusDCNet son punteros que apuntan a las compuDCNet cuyo hostname equivale a su clave en vectorCompusDCNet
- (IV) laQueMásEnvió es un puntero a la compuDCNet en vectorCompusDCNet que más paquetes enviados tiene. Si no hay compus es NULL
- (V) El conjPaquetesDCNet contiene tuplas con iteradores a todos los paquetes en tránsito en la red y sus recorridos
- (VI) Todos los paquetes en conjPaquetes de cada compuDCNet tienen id único y tanto su origen como destino existen en la topología
- (VII) El paquete en conjPaquetes tiene que tener en su recorrido a la compuDCNet en la que se encuentra y esta no puede ser igual al destino del recorrido
- (VIII) Las claves de diccPaquetesDCNet son los id de los paquetes en conjPaquetes
- (IX) Los significados de diccPaquetesDCNet son un iterador al paqueteDCNet de conjPaquetesDCNet que contiene un iterador al paquete con el id equivalente a su clave y un recorrido que es uno de los caminos mínimos del origen del paquete a la compu en la que se encuentra
- (X) La cantidad de enviados de una compuDCNet es igual o mayor a la cantidad de apariciones de esa compu en los caminos recorridos de paquetes en la red
- (XI) El paquete a enviar de cada compuDCNet es un iterador que no tiene siguiente

Rep : estr  $\rightarrow$  bool

Rep( $e$ )  $\equiv$  true  $\iff$   
 $(\forall c: \text{compu})(c \in \text{computadoras}(e.\text{topologia}) \iff$   
 $($   
 $(\exists cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \wedge (cd.pc = \text{puntero}(c)) \wedge$   
 $(\exists s: \text{string})(\text{def?}(s, e.\text{diccCompusDCNet}) \wedge (s = c.ip)))$   
 $)$   
 $) \wedge_L$   
 $(\forall cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \iff$   
 $(\exists s: \text{string})((s = cd.pc \rightarrow ip) \wedge \text{def?}(s, e.\text{diccCompusDCNet}) \wedge_L$   
 $\text{obtener}(s, e.\text{diccCompusDCNet}) = \text{puntero}(cd))$   
 $) \wedge_L$   
 $(\exists cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \wedge_L$   
 $* (cd.pc) = \text{compuQueMásEnvio}(e.\text{vectorCompusDCNet}) \wedge e.\text{laQueMásEnvio} = \text{puntero}(cd)) \wedge_L$   
 $(\forall cd_1: \text{compuDCNet})(\text{está?}(cd_1, e.\text{vectorCompusDCNet}) \Rightarrow$   
 $(\forall p_1: \text{paquete})(p_1 \in cd_1.\text{conjPaquetes} \Rightarrow$   
 $(\forall cd_2: \text{compuDCNet})(\text{está?}(cd_2, e.\text{vectorCompusDCNet}) \wedge cd_1 \neq cd_2) \Rightarrow$   
 $(\forall p_2: \text{paquete})(p_2 \in cd_2.\text{conjPaquetes} \Rightarrow p_1.id \neq p_2.id)$   
 $)$   
 $)$   
 $) \wedge_L$   
 $(\forall cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \Rightarrow$   
 $($   
 $(\forall p: \text{paquete})(p \in cd.\text{conjPaquetes} \iff$   
 $($   
 $((p.\text{origen} \in \text{computadoras}(e.\text{topologia}) \wedge p.\text{destino} \in \text{computadoras}(e.\text{topologia}) \wedge$   
 $p.\text{destino} \neq *(cd.pc)) \wedge_L$   
 $(\exists sc: \text{secu}(\text{compu}))(sc \in \text{caminosMinimos}(e.\text{topologia}, p.\text{origen}, p.\text{destino}) \wedge \text{está?}(*(cd.pc), sc))) \wedge$   
 $(\exists n: \text{nat}) ((\text{def?}(n, cd.\text{diccPaquetesDCNet}) \wedge p.id = n) \wedge_L$   
 $(\exists pdn: \text{paqueteDCNet})(pdn \in e.\text{conjPaquetesDCNet} \wedge \text{Siguiente}(pdn.it) = p \wedge$   
 $((p.\text{origen} = *(cd.pc) \wedge pdn.\text{recorrido} = *(cd.pc) \bullet <>) \vee$   
 $(p.\text{origen} \neq *(cd.pc) \wedge pdn.\text{recorrido} \in \text{caminosMinimos}(e.\text{topologia}, p.\text{origen}, *(cd.pc)))) \wedge$   
 $\text{Siguiente}(\text{obtener}(n, cd.\text{diccPaquetesDCNet})) = pdn$   
 $)$   
 $)$   
 $) \wedge_L$   
 $(\neg \text{vacía?}(cd.\text{colaPaquetesDCNet}) \iff$   
 $(\exists p: \text{paquete})(p \in cd.\text{conjPaquetes}) \wedge (p = \text{paqueteMásPrioridad}(cd.\text{conjPaquetes})) \wedge$   
 $(\exists pdn: \text{paqueteDCNet})(pdn \in e.\text{conjPaquetesDCNet}) \wedge (\text{Siguiente}(pdn.it) = p) \wedge$   
 $(\text{Siguiente}(\text{proximo}(cd.\text{colaPaquetesDCNet})) = pdn))$   
 $)$   
 $) \wedge_L$   
 $(cd.\text{enviados} \geq \text{enviadosCompu}(*(cd.pc), e.\text{vectorCompusDCNet})) \wedge$   
 $(\neg \text{HaySiguiente?}(cd.\text{paqueteAEnviar}))$   
 $)$

compuQueMásEnvio : secu(compuDCNet)  $s cd \rightarrow$  compu  $\{\neg \text{vacía?}(s cd)\}$

maxEnvio : secu(compuDCNet)  $s cd \rightarrow$  nat  $\{\neg \text{vacía?}(s cd)\}$

enviaronK : secu(compuDCNet)  $\times$  nat  $\rightarrow$  conj(compu)

paqueteMásPrioridad : conj(paquete)  $cp \rightarrow$  paquete  $\{\neg \emptyset?(cp)\}$

paquetesConPrioridadK : conj(paquete)  $\times$  nat  $\rightarrow$  conj(paquete)

altaPrioridad : conj(paquetes)  $cp \rightarrow$  nat  $\{\neg \emptyset?(cp)\}$

enviadosCompu : compu  $\times$  secu(compuDCNet)  $\rightarrow$  nat

aparicionesCompu : compu  $\times$  conj(nat)  $cn \times \text{dicc}(\text{nat} \times \text{itConj}(\text{paqueteDCNet})) dp \rightarrow$  nat

$$\{\text{claves}(dp) \subseteq cn\}$$

```

compuQueMásEnvió(scd)  $\equiv$  dameUno(enviaronK(scd, maxEnviado(scd)))
maxEnviado(scd)  $\equiv$  if vacía?(fin(scd)) then prim(scd).enviados else max(prim(scd), maxEnviado(fin(scd))) fi
enviaronK(scd, k)  $\equiv$  if vacía?(scd) then
     $\emptyset$ 
else
    if prim(scd).enviados = k then
        Ag(* (prim(scd).pc), enviaronK(fin(scd), k))
    else
        enviaronK(fin(scd), k)
    fi
fi
paqueteMásPrioridad(dcn, cp)  $\equiv$  dameUno(paquetesConPrioridadK(cp, altaPrioridad(cp)))
altaPrioridad(cp)  $\equiv$  if  $\emptyset$ ?(sinUno(cp)) then
    dameUno(cp).prioridad
else
    min(dameUno(cp).prioridad, altaPrioridad(sinUno(cp)))
fi
paquetesConPrioridadK(cp, k)  $\equiv$  if  $\emptyset$ ?(cp) then
     $\emptyset$ 
else
    if dameUno(cp).prioridad = k then
        Ag(dameUno(cp), paquetesConPrioridadK(sinUno(cp), k))
    else
        paquetesConPrioridadK(sinUno(cp), k)
    fi
fi
enviadosCompu(c, scd)  $\equiv$  if vacía?(scd) then
    0
else
    if prim(scd) = c then
        enviadosCompu(c, fin(scd))
    else
        aparicionesCompu(c, claves(prim(scd).diccPaquetesDCNet),
        prim(scd).diccPaquetesDCNet) + enviadosCompu(c, fin(scd))
    fi
fi
aparicionesCompu(c, cn, dpc)  $\equiv$  if  $\emptyset$ ?(cn) then
    0
else
    if está?(c, Siguiente(obtener(dameUno(cn), dpc)).recorrido) then
        1 + aparicionesCompu(c, sinUno(cn), dpc)
    else
        aparicionesCompu(c, sinUno(cn), dpc)
    fi
fi

```



### 1.2.3. Función de Abstracción

$\text{Abs} : \text{estr } e \longrightarrow \text{dcnet} \quad \{\text{Rep}(e)\}$   
 $\text{Abs}(e) =_{\text{obs}} \text{dcn} : \text{dcnet} \mid \text{red}(\text{dcn}) = e.\text{topología} \wedge$   
 $(\forall \text{cdn} : \text{compuDCNet})(\text{está}?(cdn, e.\text{vectorCompusDCNet}) \Rightarrow_L$   
 $\text{enEspera}(\text{dcn}, *(cdn.\text{pc})) = \text{cdn}.\text{conjPaquetes} \wedge$   
 $\text{cantidadEnviados}(\text{dcn}, *(cdn.\text{pc})) = \text{cdn}.\text{enviados} \wedge$   
 $(\forall p : \text{paquete})(p \in \text{cdn}.\text{conjPaquetes} \Rightarrow_L$   
 $\text{caminoRecorrido}(\text{dcn}, p) = \text{Siguiente}(\text{obtener}(p.\text{id}, \text{cdn}.\text{diccPaquetesDCNet})).\text{recorrido}$   
 $)$   
 $)$

### 1.3. Algoritmos

**iniciarDCNet** (**in** *topo* : **red**)  $\rightarrow$  **res** : **estr**

|  |                  |
|--|------------------|
| <code>res.topologia <math>\leftarrow</math> Copiar(topo)</code>  | $O(n! * n^6)$    |
| <code>res.vectorCompusDCNet <math>\leftarrow</math> Vacía()</code>   | $O(1)$           |
| <code>res.diccCompusDCNet <math>\leftarrow</math> CrearDicc()</code>   | $O(1)$           |
| <code>res.laQueMasEnvio <math>\leftarrow</math> NULL</code>  | $O(1)$           |
| <code>res.conjPaquetesDCNet <math>\leftarrow</math> Vacío()</code>   | $O(1)$           |
| <br><code>itConj(compu): it <math>\leftarrow</math> CrearIt(Computadoras(topo))</code>                         | $O(1)$           |
| <br><code>if (HaySiguiente?(it)) then</code>   | $O(1)$           |
| <code>res.laQueMasEnvio <math>\leftarrow</math> puntero(Siguiente(it))</code>                                  | $O(1)$           |
| <code>end if</code>  |                  |
| <br><code>while HaySiguiente?(it) do</code>  | $O(1)$           |
| <code>compuDCNet: computdcnet <math>\leftarrow</math> &lt;puntero(Siguiente(it)), Vacío(), CrearDicc(),</code> |                  |
| <code>Vacía(), CrearIt(Vacío()), 0&gt;</code>  | $O(1)$           |
| <code>AgregarAtras(res.vectorCompusDCNet, computdcnet)</code>  | $O(n)$           |
| <code>Definir(res.diccCompusDCNet, Siguiente(it).ip, puntero(computdcnet))</code>                              | $O(L)$           |
| <code>Avanzar(it)</code>   | $O(1)$           |
| <code>end while</code>   | $O(n * (n + L))$ |

**Complejidad** :  $O(n * (n + L))$

**iCrearPaquete** (**in/out** *dcn* : **dcnet**, **in** *p* : **paquete**)

|   |              |
|---|--------------|
| <code>puntero(compuDCNet): computdcnet <math>\leftarrow</math></code>   |              |
| <code>Significado(dcn.diccCompusDCNet, p.origen.ip)</code>  | $O(L)$       |
| <code>itConj(paquete): itPaq <math>\leftarrow</math> AgregarRapido(computdcnet <math>\rightarrow</math> conjPaquetes, p)</code> | $O(1)$       |
| <code>lista(compu): recorr <math>\leftarrow</math> AgregarAtras(Vacía(), p.origen)</code>                                       | $O(1)$       |
| <code>paqueteDCNet: paqDCNet <math>\leftarrow</math> &lt;itPaq, recorr&gt;</code>   | $O(1)$       |
| <br><code>itConj(paqueteDCNet): itPaqDCNet <math>\leftarrow</math></code>   |              |
| <code>AgregarRapido(dcn.conjPaquetesDCNet, paqDCNet)</code>   | $O(1)$       |
| <code>Definir(computdcnet <math>\rightarrow</math> diccPaquetesDCNet, p.id, itPaqDCNet)</code>                                  | $O(\log(k))$ |
| <code>Encolar(computdcnet <math>\rightarrow</math> colaPaquetesDCNet, p.prioridad, itPaqDCNet)</code>                           | $O(\log(k))$ |

**Complejidad** :  $O(\log(k) + L)$

iAvanzarSegundo (in/out dcn: dcnet)

```

nat: maxEnviados ← 0
nat: i ← 0
while i < Longitud(dcn.vectorCompusDCNet) do
    if (¬EsVacia?(dcn.vectorCompusDCNet[i].colaPaquetesDCNet)) then
        dcn.vectorCompusDCNet[i].paqueteAEnviar ←
            Desencolar(dcn.vectorCompusDCNet[i].colaPaquetesDCNet)
    end if
    i++
end while

i ← 0
while i < Longitud(dcn.vectorCompusDCNet) do
    if (HaySiguiente?(dcn.vectorCompusDCNet[i].paqueteAEnviar)) then

        dcn.vectorCompusDCNet[i].enviados++
        if (dcn.vectorCompusDCNet[i].enviados > maxEnviados) then
            dcn.laQueMasEnvio ← puntero(dcn.vectorCompusDCNet[i])
        end if

        paquete: pAEnviar ←
            Siguiente(Siguiente(dcn.vectorCompusDCNet[i].paqueteAEnviar).it)
        itConj(lista(compu)): intercaminos ←
            CrearIt(CaminosMinimos(dcn.topologia,
                *(dcn.vectorCompusDCNet[i].pc), pAEnviar.destino))
        compu: siguientecompu ← Siguiente(intercaminos)[1]

        if (pAEnviar.destino ≠ siguientecompu) then

            compuDCNet: siguientecompudcnet ←
                *(Obtener(dcn.diccCompusDCNet, siguientecompu.ip))

            itConj(paquete): itpaquete ←
                AgregarRapido(siguientecompudcnet.conjPaquetes, pAEnviar)

            itConj(paqueteDCNet): paqAEnviar ←
                Obtener(dcn.vectorCompusDCNet[i].diccPaquetesDCNet,
                    pAEnviar.id)

            AgregarAtras(Siguiente(paqaEnviar).recorrido, siguientecompu)

            Encolar(siguientecompudcnet.colaPaquetesDCNet,
                pAEnviar.prioridad, paqAEnviar)
            Definir(siguientecompudcnet.diccPaquetesDCNet,
                pAEnviar.id, paqAEnviar)
        end if

        Borrar(dcn.vectorCompusDCNet[i].diccPaquetesDCNet,
            Siguiente(dcn.vectorCompusDCNet[i].paqueteAEnviar→it).id)
        EliminarSiguiente(Siguiente(dcn.vectorCompusDCNet[i].paqueteAEnviar).it)
        EliminarSiguiente(dcn.vectorCompusDCNet[i].paqueteAEnviar)

        dcn.vectorCompusDCNet[i].paqueteAEnviar ← CrearIt(Vacio())

    end if
    i++

```

```
end while
```

 $O(n * (L + \log(k)))$ 

**Complejidad :**  $O(n * (L + \log(k)))$

Red (**in** *dcn*: **dcnet**)  $\rightarrow$  res: red

```
res  $\leftarrow$  dcn.topologia
```

 $O(1)$ 

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

CaminoRecorrido (**in** *dcn*: **dcnet**, **in** *p*: **paquete**)  $\rightarrow$  res: lista(compu)

```
nat: i  $\leftarrow$  0
```

 $O(1)$ 

```
while i < Longitud(dcn.vectorCompusDCNet) do
```

 $O(1)$ 

```
  if Definido?(dcn.vectorCompusDCNet[i].diccPaquetesDCNet, p.id) then
```

 $O(\log(k))$ 

```
    res  $\leftarrow$  Siguiente(Obtener(dcn.vectorCompusDCNet[i].diccPaquetesDCNet, p.id)).recorrido
```

 $O(\log(k))$ 

```
  end if
```

```
  i++
```

 $O(1)$ 

```
end while
```

 $O(n * \log(k))$ 

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

CantidadEnviados (**in** *dcn*: **dcnet**, **in** *c*: **compu**)  $\rightarrow$  res: nat

```
res  $\leftarrow$  Obtener(dcn.diccCompusDCNet, c.ip)  $\rightarrow$  enviados
```

 $O(L)$ 

**Complejidad :**  $O(L)$

EnEspera (**in** *dcn*: **dcnet**, **in** *c*: **compu**)  $\rightarrow$  res: nat

```
res  $\leftarrow$  Obtener(dcn.diccCompusDCNet, c.ip)  $\rightarrow$  conjPaquetes
```

 $O(L)$ 

**Complejidad :**  $O(L)$

PaqueteEnTransito (**in** *dcn*: **dcnet**, **in** *p*: **paquete**)  $\rightarrow$  res: bool

```
res  $\leftarrow$  false
```

```
nat: i  $\leftarrow$  0
```

 $O(1)$ 

```
while i < Longitud(dcn.vectorCompusDCNet) do
```

 $O(1)$ 

```
  if Definido?(dcn.vectorCompusDCNet[i].diccPaquetesDCNet, p.id) then
```

 $O(\log(k))$ 

```
    res  $\leftarrow$  true
```

 $O(1)$ 

```
  end if
```

```
  i++
```

 $O(1)$ 

```
end while
```

 $O(n * \log(k))$ 

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

LaQueMasEnvio (**in**  $dcn : \text{dcnet}$ )  $\rightarrow$  res: compu

res  $\leftarrow$   $*(dcn.laQueMasEnvio \rightarrow pc)$   $O(1)$

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

$\bullet =_i \bullet$  (**in**  $dcn_1 : \text{dcnet}$ , **in**  $dcn_2 : \text{dcnet}$ )  $\rightarrow$  res: bool

bool: boolTopo  $\leftarrow dcn_1.topologia = dcn_2.topologia$   $O(n + L^2)$   
 bool: boolVec  $\leftarrow dcn_1.vectorCompusDCNet = dcn_2.vectorCompusDCNet$   $O(n * k * (k + n))$   
 bool: boolConj  $\leftarrow dcn_1.conjPaquetesDCNet = dcn_2.conjPaquetesDCNet$   $O(k^3 * (k + n))$   
 bool: boolMasEnvio  $\leftarrow *(dcn_1.laQueMasEnvio) = *(dcn_2.laQueMasEnvio)$   $O(1)$

res  $\leftarrow$  boolTopo  $\wedge$  boolVec  $\wedge$  boolTrie  $\wedge$  boolConj  $\wedge$  boolMasEnvio  $O(1)$

**Complejidad** :  $O(n * k^3 * (k + n))$

$\bullet =_{compu\text{dcn}} \bullet$  (**in**  $c_1 : \text{compuDCNet}$ , **in**  $c_2 : \text{compuDCNet}$ )  $\rightarrow$  res: bool

bool: boolPC  $\leftarrow *(c_1.pc) = *(c_2.pc)$   $O(1)$   
 bool: boolConj  $\leftarrow c_1.conjPaquetes = c_1.conjPaquetes$   $O(k^2)$   
 bool: boolAVL  $\leftarrow$  true  $O(1)$   
 bool: boolCola  $\leftarrow$  true  $O(1)$   
 bool: boolPaq  $\leftarrow$  Siguiente( $c_1.paqueteAEnviar$ )  $=_{paq\text{dcn}}$  Siguiente( $c_2.paqueteAEnviar$ )  $O(n)$   
 bool: boolEnviados  $\leftarrow c_1.enviados = c_2.enviados$   $O(1)$

if boolConj then  $O(1)$   
   itConj:  $itconj_1 \leftarrow$  CrearIt( $c_1.conjPaquetes$ )  $O(1)$   
   while HaySiguiente?( $itconj_1$ ) do  $O(1)$   
     if Definido?( $c_2.diccPaquetesDCNet$ , Siguiente( $itconj_1$ )).id then  $O(\log(n))$   
       if  $\neg$ (Siguiente(Obtener( $c_1.diccPaquetesDCNet$ , Siguiente( $itconj_1$ )).id))  $O(n)$   
          $=_{paq\text{dcn}}$   
         Siguiente(Obtener( $c_1.diccPaquetesDCNet$ , Siguiente( $itconj_1$ )).id))  $O(1)$   
         then  $O(n)$   
           boolAVL  $\leftarrow$  false  $O(1)$   
         end if  
       else  
         boolAVL  $\leftarrow$  false  $O(1)$   
       end if  
       Avanzar( $itconj_1$ )  $O(1)$   
     end while  $O(n * k)$   
 end if

if EsVacia( $c_1.colaprioridad$ ) then  $O(1)$   
   if  $\neg$ EsVacia( $c_2.colaprioridad$ ) then  $O(1)$   
     boolCola  $\leftarrow$  false  $O(1)$   
   end if  
 else  
   if EsVacia( $c_1.colaprioridad$ ) then  $O(1)$   
     boolCola  $\leftarrow$  false  $O(1)$   
   else  
     if  $\neg$ (Siguiente(Proximo( $c_1.colaprioridad$ )))  $=_{paq\text{dcn}}$   
       Siguiente(Proximo( $c_2.colaprioridad$ ))) then  $O(n)$

```

                boolCola ← false
            end if
        end if
    end if

```

```

    res ← boolPC ∧ boolConj ∧ boolAVL ∧ boolCola ∧ boolPaq ∧ boolEnviados

```

**Complejidad** :  $O(k^2 + n * k) = O(k * (k + n))$

•  $=_{paqdcn}$  • (in  $p_1$  : paqueteDCNet, in  $p_2$  : paqueteDCNet,) → res: bool

```

    bool: boolPaq ← Siguiente( $p_1.it$ ) = Siguiente( $p_2.it$ )

```

```

    bool: boolRecorrido ←  $p_1.recorrido = p_2.recorrido$ 

```

```

    res ← boolPaq ∧ boolRecorrido

```

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

## 2. Módulo Red

### 2.1. Interfaz

se explica con: RED.

géneros: red.

INICIARRED()  $\rightarrow res : red$   
**Pre**  $\equiv \{true\}$   
**Post**  $\equiv \{res =_{obs} iniciarRed\}$   
**Complejidad:**  $O(1)$   
**Descripción:** Crea una red nueva

AGREGARCOMPUTADORA(**in/out**  $r : red$ , **in**  $c : compu$ )  
**Pre**  $\equiv \{(r =_{obs} r_0) \wedge ((\forall c' : compu) (c' \in computadoras(r) \Rightarrow ip(c) \neq ip(c')))\}$   
**Post**  $\equiv \{r =_{obs} agregarComputadora(r_0, c)\}$   
**Complejidad:**  $O((n * L))$   
**Descripción:** Agrega una computadora a la red  
**Aliasing:** La compu se agrega por copia

CONECTAR(**in/out**  $r : red$ , **in**  $c : compu$ , **in**  $c' : compu$ , **in**  $i : compu$ , **in**  $i' : compu$ )  
**Pre**  $\equiv \{(r =_{obs} r_0) \wedge (c \in computadoras(r)) \wedge (c' \in computadoras(r)) \wedge (ip(c) \neq ip(c')) \wedge (\neg conectadas?(r, c, c')) \wedge (\neg usaInterfaz?(r, c, i) \wedge \neg usaInterfaz?(r, c', i'))\}$   
**Post**  $\equiv \{r =_{obs} conectar(r_0, c, i, c', i')\}$   
**Complejidad:**  $O(n! * (n^4))$   
**Descripción:** Conecta dos computadoras y recalcula los caminos mínimos de la red.

COMPUTADORAS(**in**  $r : red$ )  $\rightarrow res : conj(compu)$   
**Pre**  $\equiv \{true\}$   
**Post**  $\equiv \{alias(res =_{obs} computadoras(r))\}$   
**Complejidad:**  $O(1)$   
**Descripción:** Devuelve el conjunto de computadoras de la red.  
**Aliasing:** El conjunto se da por referencia, y es modificable si y solo si la red es modificable.

CONECTADAS?(**in**  $r : red$ , **in**  $c : compu$ , **in**  $c' : compu$ )  $\rightarrow res : bool$   
**Pre**  $\equiv \{(c \in computadoras(r)) \wedge (c' \in computadoras(r))\}$   
**Post**  $\equiv \{res =_{obs} conectadas?(r, c, c')\}$   
**Complejidad:**  $O(1)$   
**Descripción:** Indica si dos computadoras de la red estan conectadas

INTERFAZUSADA(**in**  $r : red$ , **in**  $c : compu$ , **in**  $c' : compu$ )  $\rightarrow res : interfaz$   
**Pre**  $\equiv \{conectadas?(r, c, c')\}$   
**Post**  $\equiv \{res =_{obs} interfazUsada(r, c, c')\}$   
**Complejidad:**  $O(L + n)$   
**Descripción:** Devuelve la interfaz con la cual se conecta c con c'

VECINOS(**in**  $r : red$ , **in**  $c : compu$ )  $\rightarrow res : conj(compu)$   
**Pre**  $\equiv \{c \in computadoras(r)\}$   
**Post**  $\equiv \{res =_{obs} vecinos(r, c)\}$   
**Complejidad:**  $O(n^2)$   
**Descripción:** Devuelve el conjunto de computadoras conectadas con c  
**Aliasing:** Devuelve una copia de las computadoras conectadas a c

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

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

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

**Complejidad:**  $O(L + n)$

**Descripción:** Indica si la interfaz  $i$  es usada por la computadora  $c$

CAMINOSMINIMOS(**in**  $r$ : red, **in**  $c$ : compu, **in**  $c'$ : compu)  $\rightarrow res$ : conj(lista(compu))

**Pre**  $\equiv \{(c \in computadoras(r)) \wedge (c' \in computadoras(r))\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{caminosMinimos}(r, c, i))\}$

**Complejidad:**  $O(L)$

**Descripción:** Devuelve el conjunto de caminos minimos de  $c$  a  $c'$

**Aliasing:** Devuelve una referencia no modificable

HAYCAMINO?(**in**  $r$ : red, **in**  $c$ : compu, **in**  $c'$ : compu)  $\rightarrow res$ : bool

**Pre**  $\equiv \{(c \in computadoras(r)) \wedge (c' \in computadoras(r))\}$

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

**Complejidad:**  $O(L)$

**Descripción:** Indica si existe algún camino entre  $c$  y  $c'$

COPIAR(**in**  $r$ : red)  $\rightarrow res$ : red

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

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

**Complejidad:**  $O(n! * (n^6))$

**Descripción:** Devuelve una copia la red

$\bullet = \bullet$ (**in**  $r$ : red, **in**  $r'$ : red)  $\rightarrow res$ : bool

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

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

**Complejidad:**  $O(n + L^2)$

**Descripción:** Indica si  $r$  es igual a  $r'$

## 2.2. Representación

### 2.2.1. Estructura

red se representa con **estr**

donde **estr** es **tupla**(*compus*: conj(compu) ,  
                                  *dns*: diccString(nodoRed) )

donde **nodoRed** es **tupla**(*pc*: puntero(compu) ,  
                                  *caminos*: diccString(conj(lista(compu))) ,  
                                  *conexiones*: diccLineal(nat, puntero(nodoRed)) )

donde **compu** es **tupla**(*ip*: string, *interfaces*: conj(nat))

### 2.2.2. Invariante de Representación

- (I) Todas los elementos de *compus* deben tener IPs distintas.
- (II) Para cada compu, el diccionario de strings *dns* define para la clave <IP de esa compu> un **nodoRed** cuyo *pc* es puntero a esa compu.
- (III) **nodoRed.conexiones** contiene como claves todas las interfaces usadas de la compu *c* (que tienen que estar en *pc.interfaces*)
- (IV) Ningun nodo se conecta con si mismo.
- (V) Ningun nodo se conecta a otro a traves de dos interfaces distintas.
- (VI) Para cada **nodoRed** en *dns*, *caminos* tiene como claves todas las IPs de las compus de la red (**estr.comp**), y los significados corresponden a todos los caminos mínimos desde la compu *pc* hacia la compu cuya IP es clave.

Rep : *estr*  $\rightarrow$  bool

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

(( $\forall c1, c2: \text{compu}$ ) ( $c1 \neq c2 \wedge c1 \in e.\text{compus} \wedge c2 \in e.\text{compus}$ )  $\Rightarrow c1.\text{ip} \neq c2.\text{ip}$ )  $\wedge$

(( $\forall c: \text{compu}$ ) ( $c \in e.\text{compus} \Rightarrow$   
 (  $\text{def?}(c.\text{ip}, e.\text{dns}) \wedge_L \text{obtener}(c.\text{ip}, e.\text{dns}).\text{pc} = \text{puntero}(c)$  )  
 ))  $\wedge$

(( $\forall i: \text{string}, n: \text{nodoRed}$ ) (( $\text{def?}(i, e.\text{dns}) \wedge_L n = \text{obtener}(i, e.\text{dns})$ )  $\Rightarrow$   
 ( $\exists c: \text{compu}$ ) ( $c \in e.\text{compus} \wedge (n.\text{pc} = \text{puntero}(c))$ )  
 ))  $\wedge$

(( $\forall i: \text{string}, n: \text{nodoRed}$ ) (( $\text{def?}(i, e.\text{dns}) \wedge_L n = \text{obtener}(i, e.\text{dns})$ )  $\Rightarrow$   
 (( $\forall t: \text{nat}$ ) ( $\text{def?}(t, n.\text{conexiones}) \Rightarrow (t \in n.\text{pc} \rightarrow \text{interfaces})$ ))  
 ))  $\wedge$

(( $\forall i: \text{string}, n: \text{nodoRed}$ ) (( $\text{def?}(i, e.\text{dns}) \wedge_L n = \text{obtener}(i, e.\text{dns})$ )  $\Rightarrow$   
 (( $\forall t: \text{nat}$ ) ( $\text{def?}(t, n.\text{conexiones}) \Rightarrow_L (\text{obtener}(t, n.\text{conexiones}) \neq \text{puntero}(n))$ ))  
 ))  $\wedge$

(( $\forall i: \text{string}, n: \text{nodoRed}$ ) (( $\text{def?}(i, e.\text{dns}) \wedge_L n = \text{obtener}(i, e.\text{dns})$ )  $\Rightarrow$   
 (( $\forall t1, t2: \text{nat}$ ) (( $t1 \neq t2 \wedge \text{def?}(t1, n.\text{conexiones}) \wedge \text{def?}(t2, n.\text{conexiones})$ )  $\Rightarrow_L$   
 ( $\text{obtener}(t1, n.\text{conexiones}) \neq \text{obtener}(t2, n.\text{conexiones})$ )  
 ))  
 ))  $\wedge$

(( $\forall i1, i2: \text{string}, n1, n2: \text{nodoRed}$ ) ((  
 ( $\text{def?}(i1, e.\text{dns}) \wedge_L n1 = \text{obtener}(i1, e.\text{dns})$ )  $\wedge$   
 ( $\text{def?}(i2, e.\text{dns}) \wedge_L n2 = \text{obtener}(i2, e.\text{dns})$ )  
 )  $\Rightarrow$  ( $\text{def?}(i2, n1.\text{caminos}) \wedge_L \text{obtener}(i2, n1.\text{caminos}) = \text{darCaminosMinimos}(n1, n2)$ )  
 ))  
 )

|                     |   |  |
|---------------------|---|--|
| <i>vecinas</i>      | : <i>nodoRed</i>  | $\rightarrow \text{conj}(\text{nodoRed})$                        |
| <i>auxVecinas</i>   | : <i>nodoRed</i> $\times$ <i>dicc</i> ( <i>nat</i> $\times$ <i>puntero</i> ( <i>nodoRed</i> ))  | $\rightarrow \text{conj}(\text{nodoRed})$                        |
| <i>secusDeLongK</i> | : <i>conj</i> ( <i>secu</i> ( $\alpha$ )) $\times$ <i>nat</i>   | $\rightarrow \text{conj}(\text{secu}(\alpha))$                   |
| <i>longMenorSec</i> | : <i>conj</i> ( <i>secu</i> ( $\alpha$ )) <i>secus</i>  | $\rightarrow \text{nat} \quad \{\neg \emptyset?(\text{secus})\}$ |
| <i>darRutas</i>     | : <i>nodoRed</i> <i>nA</i> $\times$ <i>nodoRed</i> <i>nB</i> $\times$ <i>conj</i> ( <i>pc</i> ) $\times$ <i>secu</i> ( <i>nodoRed</i> ) | $\rightarrow \text{conj}(\text{secu}(\text{nodoRed}))$           |



$\text{darRutasVecinas} : \text{conj}(\text{pc}) \text{ vec} \times \text{nodoRed } n \times \text{conj}(\text{pc}) \times \text{secu}(\text{nodoRed}) \longrightarrow \text{conj}(\text{secu}(\text{nodoRed}))$   
 $\text{darCaminosMinimos} : \text{nodoRed } n1 \times \text{nodoRed } n1 \longrightarrow \text{conj}(\text{secu}(\text{compu}))$

$\text{vecinas}(n) \equiv \text{auxVecinas}(n, n.\text{conexiones})$   
 $\text{auxVecinas}(n, cs) \equiv \text{if } \emptyset?(cs) \text{ then } \emptyset \text{ else } \text{Ag}(\text{obtener}(\text{dameUno}(\text{claves}(cs)), cs), \text{auxVecinas}(n, \text{sinUno}(cs)))$   
 $\text{secusDeLongK}(\text{secus}, k) \equiv \text{if } \emptyset?(\text{secus}) \text{ then } \emptyset \text{ else } \text{if } \text{long}(\text{dameUno}(\text{secus})) = k \text{ then } \text{dameUno}(\text{secus}) \cup \text{secusDeLongK}(\text{sinUno}(\text{secus}), k) \text{ else } \text{secusDeLongK}(\text{sinUno}(\text{secus}), k) \text{ fi} \text{ fi}$   
 $\text{longMenorSec}(\text{secus}) \equiv \text{if } \emptyset?(\text{sinUno}(\text{secus})) \text{ then } \text{long}(\text{dameUno}(\text{secus})) \text{ else } \text{min}(\text{long}(\text{dameUno}(\text{secus})), \text{longMenorSec}(\text{sinUno}(\text{secus}))) \text{ fi}$   
 $\text{darRutas}(nA, nB, \text{rec}, \text{ruta}) \equiv \text{if } nB \in \text{vecinas}(nA) \text{ then } \text{Ag}(\text{ruta} \circ nB, \emptyset) \text{ else } \text{if } \emptyset?(\text{vecinas}(nA) - \text{rec}) \text{ then } \emptyset \text{ else } \text{darRutas}(\text{dameUno}(\text{vecinas}(nA) - \text{rec}), nB, \text{Ag}(nA, \text{rec}), \text{ruta} \circ \text{dameUno}(\text{vecinas}(nA) - \text{rec})) \cup \text{darRutasVecinas}(\text{sinUno}(\text{vecinas}(nA) - \text{rec}), nB, \text{Ag}(nA, \text{rec}), \text{ruta} \circ \text{dameUno}(\text{vecinas}(nA) - \text{rec})) \text{ fi} \text{ fi}$   
 $\text{darRutasVecinas}(\text{vecinas}, n, \text{rec}, \text{ruta}) \equiv \text{if } \emptyset?(\text{vecinas}) \text{ then } \emptyset \text{ else } \text{darRutas}(\text{dameUno}(\text{vecinas}), n, \text{rec}, \text{ruta}) \cup \text{darRutasVecinas}(\text{sinUno}(\text{vecinas}), n, \text{rec}, \text{ruta}) \text{ fi}$   
 $\text{darCaminosMinimos}(nA, nB) \equiv \text{secusDeLongK}(\text{darRutas}(nA, nB, \emptyset, <>), \text{longMenorSec}(\text{darRutas}(nA, nB, \emptyset, <>)))$

### 2.2.3. Función de Abstracción

$Abs : \text{estr } e \longrightarrow \text{red} \quad \{\text{Rep}(e)\}$   
 $Abs(e) =_{\text{obs}} r : \text{red} \mid e.\text{compus} =_{\text{obs}} \text{computadoras}(r) \wedge$   
 $((\forall c1, c2: \text{compu}, i1, i2: \text{string}, n1, n2: \text{nodoRed}) ($   
 $(c1 \in e.\text{compus} \wedge i1 = c1.\text{ip} \wedge \text{def?}(i1, e.\text{dns}) \wedge_L n1 = \text{obtener}(i1, e.\text{dns}) \wedge c1 = *n1.\text{pc}) \wedge$   
 $(c2 \in e.\text{compus} \wedge i2 = c2.\text{ip} \wedge \text{def?}(i2, e.\text{dns}) \wedge_L n2 = \text{obtener}(i2, e.\text{dns}) \wedge c2 = *n2.\text{pc}) \wedge$   
 $(c1 \neq c2)) \Rightarrow_L$   
 $(\text{conectadas?}(r, c1, c2) \Leftrightarrow ( (\exists t1, t2: \text{nat}) ($   
 $t1 = \text{interfazUsada}(r, c1, c2) \wedge t2 = \text{interfazUsada}(r, c2, c1) \wedge$   
 $\text{def?}(t1, n1.\text{conexiones}) \wedge \text{def?}(t2, n2.\text{conexiones}) \wedge_L ($   
 $\&n2 = \text{obtener}(t1, n1.\text{conexiones}) \wedge \&n1 = \text{obtener}(t2, n2.\text{conexiones})$   
 $))))$

## 2.3. Algoritmos

```

iIniciarRed () → res: red
    res.compus ← Vacio()
    res.dns ← Vacio()
Complejidad : O(1)

```

```

iAgregarComputadora (in/out r: red, in c: compu)
    itCompus: itConj(compu) ← CrearIt(r.compus)
    while HaySiguiente?(itCompus) do
        nr: nodoRed ← Significado(r.dns, Siguiente(itCompus).ip)
        Definir(nr.camino, c.ip, Vacio())
        Avanzar(itCompus)
    end while

    AgregarRapido(r.compus, c)

    Definir(r.dns, compu.ip, Tupla<x, Vacio(), Vacio()>)
    InicializarConjCaminos(r, c)
Complejidad : O(n * L)

```

```

InicializarConjCaminos (in/out r: red, in c: compu)
    itCompus: itConj(compu) ← CrearIt(r.compus)
    cams: diccTrie(ip, conj(lista(compu))) ←
        Significado(r.dns, c.ip).camino
    while HaySiguiente?(itCompus) do
        Definir(cams, Siguiente(itCompus).ip, Vacio())
        Avanzar(itCompus)
    end while
Complejidad : O(n * L)

```

```

iConectar (in/out r: red, in c0: compu, in c1: compu, in i0: compu, in i1: compu)
  nr0:nodoRed ← Significado(r.dns, c0.ip)                                O(L)
  nr1:nodoRed ← Significado(r.dns, c1.ip)                                O(L)
  DefinirRapido(nr0.conexiones, i0, nr1)                                O(1)
  DefinirRapido(nr1.conexiones, i1, nr0)                                O(1)
  CrearTodosLosCaminos(r)                                              O(n! * (n3 * (n + L)))
Complejidad : O(n! * (n3 * (n + L)))

```

```

CrearTodosLosCaminos (in/out r: red)
  itCompuA:itConj(compu) ← CrearIt(r.compus)                            O(1)
  while HaySiguiente?(itCompuA) do                                     O(1)
    nr:nodoRed ← Significado(r.dns, Siguiente(itCompuA).ip)           O(L)

    itCompuB:itConj(compu) ← CrearIt(r.compus)                            O(1)
    while HaySiguiente?(itCompuB) do                                     O(1)

      caminimos:conj(lista(compu)) ← Minimos(Caminos
        (nr, Siguiente(itCompuB).ip)                                     O(n! * n*(n + L))
        Definir(nr.caminos, Siguiente(itCompuB).ip, caminimos)         O(L)

      Avanzar(itCompuB)                                                  O(1)
    end while                                                            O(n! * (n2 * (n + L)))

    Avanzar(itCompuA)                                                    O(1)
  end while                                                            O(n! * (n3 * (n + L)))
Complejidad : O(n! * (n3 * (n + L)))

```

```

Caminos (in c1: nodoRed, in ipDestino: string) → res: conj(lista(compu))
  res ← Vacio()                                                        O(1)

  frameRecorrido:pila(lista(compu)) ← Vacía()                          O(1)
  frameCandidatos:pila(lista(nodoRed)) ← Vacía()                       O(1)

  iCandidatos:lista(nodoRed) ← listaNodosVecinos(c1)                  O(n)
  iRecorrido:lista(compu) ← Vacía()                                     O(1)
  AgregarAdelante(iRecorrido, *(c1.pc))                              O(1)

  Apilar(frameRecorrido, iRecorrido)                                    O(1)
  Apilar(frameCandidatos, iCandidatos)                                  O(1)

  pCandidatos:compu                                                     O(1)
  fCandidatos:lista(nodoRed)                                            O(1)

  while ¬EsVacía?(frameRecorrido) do                                    O(1)
    iRecorrido ← Tope(frameRecorrido)                                   O(1)
    iCandidatos ← Tope(frameCandidatos)                                 O(1)

    Desapilar(frameRecorrido)                                            O(1)
    Desapilar(frameCandidatos)                                           O(1)

    pCandidatos ← Primero(iCandidatos)                                    O(1)

  // ... sigue

```

```

if ¬EsVacio?(iCandidatos) then                                O(1)
  Fin(iCandidatos)                                             O(1)
  fCandidatos ← iCandidatos                                    O(n)

  if ult(iRecorrido).pc→ip = ipDestino then                    O(L)
    AgregarRapido(res, iRecorrido)                             O(n)
  else
    Apilar(frameRecorrido, iRecorrido)                         O(1)
    Apilar(frameCandidatos, fCandidatos)                       O(1)

    if ¬nodoEnLista(pCandidatos, iRecorrido) then              O(n*(n + L))
      iRecorrido ← Copiar(iRecorrido)                          O(n)
      AgregarAtras(iRecorrido, *(pCandidatos))                O(n)
      Apilar(frameRecorrido, iRecorrido)                       O(1)
      Apilar(frameCandidatos, listaNodosVecinos(pCandidatos)) O(n)
    fi
  fi
fi
fi
fi
end while
Complejidad :  $O(n! * n * (n + L))$ 

```

```

Minimos (in caminos: conj(lista(compu))) → res: conj(lista(compu))
res ← Vacio()                                                  O(1)
longMinima: int                                                O(1)
itCaminos: itConj(lista(compu)) ← CrearIt(caminos)             O(1)
if HaySiguiente?(itCaminos) then                                O(1)
  longMinima ← Longitud(Siguiente(itCaminos))                 O(1)
  Avanzar(itCaminos)                                           O(1)
  while HaySiguiente?(itCaminos)                               O(1)
    if Longitud(Siguiente(itCaminos)) < longMinima then        O(1)
      longMinima ← Longitud(Siguiente(itCaminos))              O(1)
      Avanzar(itCaminos)                                       O(1)
    end while
  itCaminos ← CrearIt(caminos)                                  O(1)
  while HaySiguiente?(itCaminos)                               O(1)
    if Longitud(Siguiente(itCaminos)) = longMinima then        O(1)
      AgregarRapido(res, Siguiente(itCaminos))                 O(1)
      Avanzar(itCaminos)                                       O(1)
    end while
  end if
Complejidad :  $O(n)$ 

```

```

listaNodosVecinos (in n: nodoRed) → res: lista(nodoRed)
res ← Vacía()                                                  O(1)
itVecinos : itDicc(interfaz, puntero(nodoRed))) ← CrearIt(n, conexiones) O(1)
while HaySiguiente?(itVecinos) do                               O(1)
  AgregarAdelante(res, *SiguienteSignificado(itVecinos))      O(1)
  Avanzar(itVecinos)                                           O(1)
end while
Complejidad :  $O(n)$ 

```

```

nodoEnLista (in n: nodoRed, in ns: lista(nodoRed)) → res: bool
  res ← false
  itNodos: itLista(lista(nodoRed)) ← CrearIt(ns)
  while HaySiguiente?(itNodos) do
    if Siguiente(itNodos) = n then
      res ← true
    end if
    Avanzar(itNodos)
  end while
Complejidad :  $O(n * (n + L))$ 

```

$O(1)$   
 $O(1)$   
 $O(1)$   
 $O(n + L)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(n * (n + L))$

```

iComputadoras (in r: red) → res: conj(compu)
  res ← r.compus
Complejidad :  $O(1)$ 

```

$O(1)$

```

iConectadas? (in r: red, in c0: compu, in c1: compu) → res: bool
  nr0:nodoRed ← Significado(r.dns, c0.ip)
  it :itDicc(interfaz, puntero(nodoRed)) ← CrearIt(nr0.conexiones)
  res ← false
  while HaySiguiente?(it) do
    if c1.ip = SiguienteSignificado(it)→pc→ip then
      res ← true
    end if
    Avanzar(it)
  end while
Complejidad :  $O(L + n)$ 

```

$O(L)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(n)$

```

iInterfazUsada (in r: red, in c0: compu, in c1: compu) → res: interfaz
  nr0:nodoRed ← Significado(r.dns, c0.ip)
  it :itDicc(interfaz, puntero(nodoRed))
  ← CrearIt(nr0.conexiones)
  while HaySiguiente?(it) do
    if c1.ip = SiguienteSignificado(it)→pc→ip then
      res ← SiguienteClave(it)
    end if
    Avanzar(it)
  end while
Complejidad :  $O(L + n)$ 

```

$O(L)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(n)$

```

iVecinos (in r: red, in c: compu) → res: conj(compu)
  nr:nodoRed ← Significado(r.dns, c.ip)
  res:conj(compu) ← Vacio()
  it :itDicc(interfaz, puntero(nodoRed))
  ← CrearIt(nr.conexiones)
  while HaySiguiente?(it) do
    AgregarRapido(res, *(SiguienteSignificado(it)→pc))
    Avanzar(it)
  end while
Complejidad :  $O(L + n)$ 

```

$O(L)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(1)$   
 $O(n)$

```

iUsaInterfaz? (in r: red, in c: compu, in i: interfaz) → res: bool
  nr:nodoRed ← Significado(r.dns, c.ip)                                O(L)
  res ← Definido?(pnr.conexiones, i)                                O(n)
Complejidad :  $O(L + n)$ 

```

```

iCaminosMinimos (in r: red, in c0: compu, in c1: compu) → res: conj(secu(compu))
  nr:nodoRed ← Significado(r.dns, c0.ip)                                O(L)
  res ← Significado(pnr.caminos, c1.ip)                                O(L)
Complejidad :  $O(L)$ 

```

```

HayCamino? (in r: red, in c0: compu, in c1: compu) → res: bool
  nr:nodoRed ← Significado(r.dns, c0.ip)                                O(L)
  res ← ¬EsVacio?(Significado(pnr.caminos, c1.ip))                    O(L)
Complejidad :  $O(L)$ 

```

```

iCopiar (in otraRed: red) → res: red
  res ← iIniciarRed                                                    O(1)
  // copia el conjunto de tuplas
  res.compus ← Copiar(otraRed.compus)                                  O(n)
  // rearma los nodos (con conexiones en blanco) del diccionario dns
  itCompus:itConj(compu) ← CrearIt(res.compus)                        O(1)
  while HaySiguiente?(itCompus) do                                    O(1)

    c:compu ← Siguiente(itCompus)                                      O(1)
    nodoAux:nodoRed ← Obtener(otraRed.dns, c.ip)                       O(L)
    copiaCaminos:diccString(conj(lista(compu))) ← Copiar(nodoAux.caminos) O(n)
    Definir(res.dns, c.ip, Tupla<&c, copiaCaminos, Vacio()>)           O(L)

    Avanzar(itCompus)                                                  O(1)
  end while                                                            O(n2 + n*L)

  // rearma las conexiones
  itCompus:itConj(compu) ← CrearIt(res.compus)                        O(1)
  while HaySiguiente?(itCompus) do                                    O(1)
    nodoMio:nodoRed ← Obtener(res.dns, c.ip)                           O(L)
    nodoOtra:nodoRed ← Obtener(otraRed.dns, c.ip)                       O(L)

    itInterfs:itConj(nat) ← CrearIt(nodoMio.pc → interfaces)           O(1)

    while HaySiguiente?(itInterfs) do                                  O(1)
      interf:nat ← Siguiente(itInterfs)                                  O(1)
      ip:string ← Obtener(nodoOtra.conexiones, interf)                  O(n)
      Definir(nodoMio.conexiones, interf, &Obtener(res.dns, ip))        O(L)

      Avanzar(itInterfs)                                                O(1)
    end while                                                            O(n2 + n*L)

    Avanzar(itCompus)                                                  O(1)
  end while                                                            O(n3 + n2 * L)

Complejidad :  $O(n^3 + n^2 * L)$ 

```

$\bullet = \bullet$  (**in**  $r_0 : \text{red}$ , **in**  $r_1 : \text{red}$ )  $\rightarrow$  res: bool  
res  $\leftarrow$  ( $r_0.\text{compus} = r_1.\text{compus}$ )  $\wedge$  ( $r_0.\text{dns} = r_1.\text{dns}$ )  
**Complejidad** :  $O(n + L(L + n))$

 $O(n + L^2)$

### 3. Módulo Cola de mínima prioridad( $\alpha$ )

El módulo cola de mínima prioridad consiste en una cola de prioridad de elementos del tipo  $\alpha$  cuya prioridad está determinada por un *nat* de forma tal que el elemento que se ingrese con el menor *nat* será el de mayor prioridad.

#### 3.1. Especificación

**TAD COLA DE MÍNIMA PRIORIDAD( $\alpha$ )**

**igualdad observacional**

$$(\forall c, c' : \text{colaMinPrior}(\alpha)) \left( c =_{\text{obs}} c' \iff \left( \begin{array}{l} \text{vacía?}(c) =_{\text{obs}} \text{vacía?}(c') \wedge_{\text{L}} \\ (\neg \text{vacía?}(c) \Rightarrow_{\text{L}} (\text{próximo}(c) =_{\text{obs}} \text{próximo}(c') \wedge \\ \text{desencolar}(c) =_{\text{obs}} \text{desencolar}(c')) \end{array} \right) \right)$$

**parámetros formales**

**géneros**  $\alpha$

**operaciones**  $\bullet < : \alpha \times \alpha \rightarrow \text{bool}$

Relación de orden total estricto<sup>1</sup>

$\bullet$

**géneros**  $\text{colaMinPrior}(\alpha)$

**exporta**  $\text{colaMinPrior}(\alpha)$ , generadores, observadores

**usa** **BOOL**

**observadores básicos**

$\text{vacía?} : \text{colaMinPrior}(\alpha) \rightarrow \text{bool}$

$\text{próximo} : \text{colaMinPrior}(\alpha) \ c \rightarrow \alpha$

$\{\neg \text{vacía?}(c)\}$

$\text{desencolar} : \text{colaMinPrior}(\alpha) \ c \rightarrow \text{colaMinPrior}(\alpha)$

$\{\neg \text{vacía?}(c)\}$

**generadores**

$\text{vacía} : \rightarrow \text{colaMinPrior}(\alpha)$

$\text{encolar} : \alpha \times \text{colaMinPrior}(\alpha) \rightarrow \text{colaMinPrior}(\alpha)$

**otras operaciones**

$\text{tamaño} : \text{colaMinPrior}(\alpha) \rightarrow \text{nat}$

**axiomas**  $\forall c : \text{colaMinPrior}(\alpha), \forall e : \alpha$

$\text{vacía?}(\text{vacía}) \equiv \text{true}$

$\text{vacía?}(\text{encolar}(e, c)) \equiv \text{false}$

$\text{próximo}(\text{encolar}(e, c)) \equiv \text{if } \text{vacía?}(c) \vee_{\text{L}} \text{próximo}(c) > e \text{ then } e \text{ else } \text{próximo}(c) \text{ fi}$

$\text{desencolar}(\text{encolar}(e, c)) \equiv \text{if } \text{vacía?}(c) \vee_{\text{L}} \text{próximo}(c) > e \text{ then } c \text{ else } \text{encolar}(e, \text{desencolar}(c)) \text{ fi}$

**Fin TAD**

<sup>1</sup>Una relación es un orden total estricto cuando se cumple:

**Antirreflexividad:**  $\neg a < a$  para todo  $a : \alpha$

**Antisimetría:**  $(a < b \Rightarrow \neg b < a)$  para todo  $a, b : \alpha, a \neq b$

**Transitividad:**  $((a < b \wedge b < c) \Rightarrow a < c)$  para todo  $a, b, c : \alpha$

**Totalidad:**  $(a < b \vee b < a)$  para todo  $a, b : \alpha$



### 3.2. Interfaz

**parámetros formales**  
**géneros**       $\alpha$

**se explica con:** COLA DE MÍNIMA PRIORIDAD(NAT).

**géneros:** colaMinPrior( $\alpha$ ).

#### 3.2.1. Operaciones básicas de Cola de mínima prioridad

VACÍA()  $\rightarrow res : \text{colaMinPrior}(\alpha)$

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

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

**Complejidad:** O(1)

**Descripción:** Crea una cola de prioridad vacía

VACÍA?(in  $c : \text{colaMinPrior}(\alpha)$ )  $\rightarrow res : \text{bool}$

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

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

**Complejidad:** O(1)

**Descripción:** Devuelve true si y sólo si la cola está vacía

PRÓXIMO(in  $c : \text{colaMinPrior}(\alpha)$ )  $\rightarrow res : \alpha$

**Pre**  $\equiv \{\neg \text{vacía?}(c)\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{próximo}(c))\}$

**Complejidad:** O(1)

**Descripción:** Devuelve el próximo elemento a desencolar

**Aliasing:**  $res$  es modificable si y sólo si  $c$  es modificable

DESENCOLAR(in/out  $c : \text{colaMinPrior}(\alpha)$ )  $\rightarrow res : \alpha$

**Pre**  $\equiv \{\neg \text{vacía?}(c) \wedge c =_{\text{obs}} c_0\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{próximo}(c_0) \wedge c =_{\text{obs}} \text{desencolar}(c_0)\}$

**Complejidad:** O(log(tamaño( $c$ )))

**Descripción:** Quita el elemento más prioritario

**Aliasing:** Se devuelve el elemento por copia

ENCOLAR(in/out  $c : \text{colaMinPrior}(\alpha)$ , in  $p : \text{nat}$ , in  $a : \alpha$ )

**Pre**  $\equiv \{c =_{\text{obs}} c_0\}$

**Post**  $\equiv \{c =_{\text{obs}} \text{encolar}(p, c_0)\}$

**Complejidad:** O(log(tamaño( $c$ )))

**Descripción:** Agrega al elemento  $\alpha$  con prioridad  $p$  a la cola

**Aliasing:** Se agrega el elemento por copia

• = •(in  $c : \text{colaMinPrior}(\alpha)$ , in  $c' : \text{colaMinPrior}(\alpha)$ )  $\rightarrow res : \text{bool}$

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

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

**Complejidad:** O(min(tamaño( $c$ ), tamaño( $c'$ )))

**Descripción:** Indica si  $c$  es igual  $c'$

### 3.3. Representación

#### 3.3.1. Representación de colaMinPrior

colaMinPrior( $\alpha$ ) se representa con **estr**

donde **estr** es **diccLog**(nodoEncolados)

donde **nodoEncolados** es **tupla**(*encolados*: cola( $\alpha$ ), *prioridad*: nat)

#### 3.3.2. Invariante de Representación

- (I) Todos los significados del diccionario tienen como clave el valor de *prioridad*
- (II) Todos los significados del diccionario no pueden tener una cola vacía

Rep : **estr**  $\rightarrow$  bool

Rep( $e$ )  $\equiv$  true  $\iff$   
 $(\forall n : \text{nat}) \text{ def?}(n, e) \Rightarrow_L ((\text{obtener}(n, e).\text{prioridad} = n) \wedge \neg \text{vacía?}(\text{obtener}(n, e).\text{encolados}))$

#### 3.3.3. Función de Abstracción

Abs : **estr**  $e \rightarrow$  colaMinPrior

{Rep( $e$ )}

Abs( $e$ ) =<sub>obs</sub> cmp: colaMinPrior | (vacía?(*cmp*)  $\Leftrightarrow$  (#claves( $e$ ) = 0))  $\wedge$   
 $\neg \text{vacía?}(\text{cmp}) \Rightarrow_L$   
 $((\text{próximo}(\text{cmp}) = \text{próximo}(\text{mínimo}(e).\text{encolados})) \wedge$   
 $(\text{desencolar}(\text{cmp}) = \text{desencolar}(\text{mínimo}(e).\text{encolados})))$

### 3.4. Algoritmos

iVacía ()  $\rightarrow$  res: colaMinPrior( $\alpha$ )

res  $\leftarrow$  CrearDicc()

O(1)

**Complejidad** : O(1)

iVacía? (**in**  $c$ : colaMinPrior( $\alpha$ ))  $\rightarrow$  res: bool

res  $\leftarrow$  Vacío?( $c$ )

O(1)

**Complejidad** : O(1)

iPróximo (**in/out**  $c$ : colaMinPrior( $\alpha$ ))  $\rightarrow$  res:  $\alpha$

res  $\leftarrow$  Proximo(Minimo( $c$ )).encolados)

O(1)

**Complejidad** : O(1)

iDesencolar (**in/out**  $c$ : colaMinPrior( $\alpha$ ))  $\rightarrow$  res:  $\alpha$

```

res  $\leftarrow$  Copiar(Proximo(Minimo( $c$ ).encolados))           O(copy( $\alpha$ ))
Desencolar(Minimo( $c$ ).encolados)                         O(log(tamaño( $c$ )))
if EsVacia?(Minimo( $c$ ).encolados) then                   O(1)
    Borrar( $c$ , Minimo( $c$ ).prioridad)                       O(log(tamaño( $c$ )))
end if

```

**Complejidad** :  $O(\log(\text{tamano}(c)) + O(\text{copy}(\alpha)))$

iEncolar (**in/out**  $c$ : colaMinPrior( $\alpha$ ), **in**  $p$ : nat, **in**  $a$ :  $\alpha$ )

```

if Definido?( $c$ ,  $p$ ) then                                   O(log(tamaño( $c$ )))
    Encolar(Obtener( $c$ ,  $p$ ).encolados,  $a$ )                 O(log(tamaño( $c$ )) + copy( $\alpha$ ))
else
    nodoEncolados nuevoNodoEncolados                     O(1)
    nuevoNodoEncolados.encolados  $\leftarrow$  Vacía()         O(1)
    nuevoNodoEncolados.prioridad  $\leftarrow$   $p$                O(1)
    Encolar(nuevoNodoEncolados.encolados,  $a$ )             O(copy( $a$ ))
    Definir( $c$ ,  $p$ , nuevoNodoEncolados)                   O(log(tamaño( $c$ )) + copy(nodoEncolados))
end if

```

**Complejidad** :  $O(\log(\text{tamano}(c)) + O(\text{copy}(\alpha)))$

$\bullet = \bullet$  (**in**  $c_0$ : colaMinPrior( $\alpha$ ), **in**  $c_1$ : colaMinPrior( $\alpha$ ))  $\rightarrow$  res: bool

```

res  $\leftarrow$   $c_0 = c_1$                                      O(min(tamano( $c_0$ ), tamaño( $c_1$ )))

```

**Complejidad** :  $O(\min(\text{tamano}(c_0), \text{tamano}(c_1)))$

## 4. Módulo Diccionario logarítmico( $\alpha$ )

### 4.1. Interfaz

se explica con:  $\text{DICCIONARIO}(\text{NAT}, \alpha)$ .

géneros:  $\text{diccLog}(\alpha)$ .

#### 4.1.1. Operaciones básicas de Diccionario logarítmico( $\alpha$ )

$\text{CREARDICC}() \rightarrow res : \text{diccLog}(\alpha)$

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

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

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

**Descripción:** Crea un diccionario vacío

$\text{DEFINIDO?}(\text{in } d : \text{diccLog}(\alpha), \text{in } c : \text{nat}) \rightarrow res : \text{bool}$

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

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

**Complejidad:**  $O(\log(\#claves(d)))$

**Descripción:** Devuelve **true** si y sólo si la clave fue previamente definida en el diccionario

$\text{DEFINIR}(\text{in/out } d : \text{diccLog}(\alpha), \text{in } c : \text{nat}, \text{in } s : \alpha)$

**Pre**  $\equiv \{d =_{\text{obs}} d_0\}$

**Post**  $\equiv \{d =_{\text{obs}} \text{definir}(c, s, d_0)\}$

**Complejidad:**  $O(\log(\#claves(d)) + \text{copy}(s))$

**Descripción:** Define la clave  $c$  con el significado  $s$  en  $d$

$\text{OBTENER}(\text{in/out } d : \text{diccLog}(\alpha), \text{in } c : \text{nat}) \rightarrow res : \alpha$

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

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

**Complejidad:**  $O(\log(\#claves(d)))$

**Descripción:** Devuelve el significado correspondiente a la clave en el diccionario

**Aliasing:**  $res$  es modificable si y sólo si  $d$  es modificable

$\text{BORRAR}(\text{in/out } d : \text{diccLog}(\alpha), \text{in } c : \text{nat})$

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

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

**Complejidad:**  $O(\log(\#claves(d)))$

**Descripción:** Borra el elemento con la clave dada

$\text{VACÍO?}(\text{in } d : \text{diccLog}(\alpha)) \rightarrow res : \text{bool}$

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

**Post**  $\equiv \{res =_{\text{obs}} \emptyset?(claves(d))\}$

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

**Descripción:** Devuelve **true** si y sólo si el diccionario está vacío

$\text{MÍNIMO}(\text{in/out } d : \text{diccLog}(\alpha)) \rightarrow res : \alpha$

**Pre**  $\equiv \{\neg \emptyset?(claves(d))\}$

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

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

**Descripción:** Devuelve el significado correspondiente a la clave de mínimo valor en el diccionario

**Aliasing:**  $res$  es modificable si y sólo si  $d$  es modificable

**•** = **•**(in  $d: \text{diccLog}(\alpha)$ , in  $d': \text{diccLog}(\alpha)$ )  $\rightarrow res: \text{bool}$   
**Pre**  $\equiv \{\text{true}\}$   
**Post**  $\equiv \{res =_{\text{obs}} (d =_{\text{obs}} d')\}$   
**Complejidad:**  $O(\max(\#claves(d), \#claves(d')))$   
**Descripción:** Devuelve **true** si y sólo si ambos diccionarios son iguales

#### 4.1.2. Operaciones auxiliares del TAD

$\text{claveMínima} : \text{dicc}(\text{nat} \times \alpha) \ d \longrightarrow \text{nat} \quad \{\neg \emptyset?(claves(d))\}$   
 $\text{darClaveMínima} : \text{dicc}(\text{nat} \times \alpha) \ d \times \text{conj}(\text{nat}) \ c \longrightarrow \text{nat} \quad \{\neg \emptyset?(claves(d)) \wedge (c \subseteq claves(d))\}$   
 $\text{claveMínima}(d) \equiv \text{darClaveMínima}(d, claves(d))$   
 $\text{darClaveMínima}(d, c) \equiv \text{if } \emptyset?(\text{sinUno}(c)) \text{ then } \text{dameUno}(c) \text{ else } \text{min}(\text{dameUno}(c), \text{darClaveMínima}(d, \text{sinUno}(c))) \text{ fi}$

## 4.2. Representación

### 4.2.1. Representación de $\text{diccLog}(\alpha)$

$\text{diccLog}(\alpha)$  se representa con **estr**

donde **estr** es  $\text{tupla}(\text{abAvl}: \text{ab}(\text{nodoAvl}), \text{mínimo}: \text{puntero}(\text{ab}(\text{nodoAvl})))$   
 donde **nodoAvl** es  $\text{tupla}(\text{clave}: \text{nat}, \text{data}: \alpha, \text{balance}: \text{int})$

### 4.2.2. Invariante de Representación

- (I) Se mantiene el invariante de árbol binario de búsqueda para las claves de los nodos.
- (II) Cada nodo tiene  $\text{balance} \in \{-1, 0, 1\}$  donde *balance* es:
  - \* 0 si el árbol está balanceado
  - \* 1 si la diferencia en altura entre el hijo derecho y el izquierdo es de uno
  - \* -1 si la diferencia en altura entre el hijo izquierdo y el derecho es de uno
- (III) Todas las claves son distintas.
- (IV) El *mínimo* apunta al árbol con la clave de menor valor, si el diccionario está vacío vale **NULL**.

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

$\text{Rep}(e) \equiv \text{true} \iff \text{esABB}(e.\text{abAvl}) \wedge \text{balanceadoBien}(e.\text{abAvl}) \wedge \text{clavesÚnicas}(e.\text{abAvl}, \text{vacío}) \wedge_L$   
 $e.\text{mínimo} = \text{árbolClaveMínima}(e.\text{abAvl})$

$\text{esABB} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{bool}$   
 $\text{balanceadoBien} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{bool}$   
 $\text{clavesEnÁrbol} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{conj}(\text{nat})$   
 $\text{clavesÚnicas} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{bool}$   
 $\text{árbolClaveMínima} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{puntero}(\text{ab}(\text{nodoAvl}))$   
 $\text{darSignificado} : \text{ab}(\text{nodoAvl}) \ a \times \text{nat} \ c \longrightarrow \alpha \quad \{c \in \text{clavesEnÁrbol}(a) \wedge \text{esABB}(a)\}$

```

esABB(a)           ≡ (¬nil?(a)) ⇒L (
    (¬nil?(izq(a)) ⇒L (raíz(a).clave > raíz(izq(a)).clave ∧ esABB(izq(a)))) ∧
    (¬nil?(der(a)) ⇒L (raíz(a).clave < raíz(der(a)).clave ∧ esABB(der(a))))
)

balanceadoBien(a)  ≡ if (nil?(a)) then
    true
else
    (abs(altura(der(a)) - altura(izq(a))) < 2) ∧
    (raíz(a)→balance = altura(der(a)) - altura(izq(a))) ∧
    balanceadoBien(izq(a)) ∧ balanceadoBien(der(a))
fi

clavesEnÁrbol(a)   ≡ if (nil?(a)) then
    ∅
else
    Ag(raíz(a).clave, (clavesEnÁrbol(izq(a)) ∪ clavesEnÁrbol(der(a))))
fi

clavesÚnicas(a)    ≡ tamaño(a) = #(clavesEnÁrbol(a))

árbolClaveMínima(a) ≡ if (nil?(a)) then
    NULL
else
    if (nil?(izq(a))) then puntero(a) else árbolClaveMínima(izq(a)) fi
fi

darSignificado(a, c) ≡ if (raíz(a).clave = c) then
    raíz(a).data
else
    if (raíz(a).clave < c) then
        darSignificado(izq(a), c)
    else
        darSignificado(der(a), c)
    fi
fi

```

#### 4.2.3. Función de Abstracción

```

Abs : estr e  → dicc(nat, α)                                     {Rep(e)}
Abs(e) =obs d: dicc(nat, α) | (∀n: nat) (
    (def?(n, d) ⇔ n ∈ clavesEnÁrbol(e.abAvl)) ∧L
    (def?(n, d) ⇒L obtener(n, d) = darSignificado(e.abAvl, n))
)

```

### 4.3. Algoritmos

|  |      |
|--|------|
| <pre> iCrearDicc () → res: diccLog(α)   res ← &lt;Nil, NULL&gt; <b>Complejidad</b> : O(1) </pre> | O(1) |
|--|------|

|  |                    |
|--|--------------------|
| <pre> iDefinir (<b>in/out</b> diccLog(α): d, <b>in</b> nat: c, <b>in</b> α: s)   <b>if</b> (Nil?(d.abAvl)) <b>then</b>     d.abAvl ← crearArbol(c, s) </pre> | O(1)<br>O(copy(s)) |
|--|--------------------|

```

    d.minimo ← puntero(d.abAvl)                                O(1)
else
    it: ab(nodoAvl) ← d.abAvl                                  O(1)
    up: pila(puntero(ab(nodoAvl)))                             O(1)
    upd: pila(bool)                                             O(1)

    Apilar(upd, (Raiz(it).clave < c))                           O(1)
    Apilar(up, puntero(it))                                     O(1)

    while(¬Nil?(subArbol(it, Tope(upd))))                       O(1)
        it ← subArbol(it, Tope(upd))                           O(1)

        Apilar(upd, (Raiz(it).clave < c))                       O(1)
        Apilar(up, puntero(it))                                 O(1)
    do                                                         O(log(#claves(d)))

    subArbol(it, Tope(upd)) ← crearArbol(c, s)                  O(copy(s))
    if(c < Raiz(*d.minimo).clave) then                          O(1)
        d.minimo ← puntero(subArbol(it, Tope(upd)))            O(1)
    end if

    break ← false                                              O(1)
    while((Tamano(up) > 0) ∧ ¬break)                             O(1)
        if(Tope(upd)) then                                     O(1)
            Raiz(*Tope(up)).balance ← Raiz(*Tope(up)).balance + 1 O(1)
        else
            Raiz(*Tope(up)).balance ← Raiz(*Tope(up)).balance - 1 O(1)
        end if

        if(Raiz(*Tope(up)).balance = 0) then                    O(1)
            break ← true                                         O(1)
        else
            if(abs(Raiz(*Tope(up)).balance > 1)) then           O(1)
                *Tope(up) ← puntero(insertarBalance(*Tope(up), Tope(upd))) O(1)

                if(Tamano(up) > 1) then                           O(1)
                    upTope: puntero(ab(nodoAvl)) ← copy(Tope(up)) O(1)
                    Desapilar(up)                               O(1)
                    Desapilar(upd)                               O(1)
                    subArbol(*Tope(up), Tope(upd)) ← *upTope    O(1)
                else
                    d.abAvl ← *Tope(up)                          O(1)
                end if

                break ← true                                     O(1)
            else
                Desapilar(up)                                    O(1)
                Desapilar(upd)                                    O(1)
            end if
        end if
    do                                                         O(log(#claves(d)))
end if
Complejidad :  $O(\log(\#claves(d))) + O(copy(s))$ 

```

```

crearArbol (in nat: c, in α: s) → res: ab(nodoAvl)
res ← Bin(Nil, <c, copy(s), 0>, Nil)
Complejidad :  $O(copy(s))$ 

```

O(copy(s))

```

subArbol (in/out  $ab(nodoAvl): a$ , in  $bool: dir$ )  $\rightarrow$  res:  $ab(nodoAvl)$ 
  if( $dir$ ) then
    res  $\leftarrow$  Der( $a$ )
  else
    res  $\leftarrow$  Izq( $a$ )
  end if
Complejidad :  $O(1)$ 

```

```

insertarBalance (in/out  $ab(nodoAvl): root$ , in  $bool: dir$ )  $\rightarrow$  res:  $ab(nodoAvl)$ 
  hijo:  $ab(nodoAvl) \leftarrow subArbol(root, dir)$ 

  if( $dir$ ) then
    bal:  $int \leftarrow 1$ 
  else
    bal:  $int \leftarrow -1$ 
  end if

  if( $Raiz(hijo).balance = bal$ ) then
     $Raiz(root).balance \leftarrow 0$ 
     $Raiz(hijo).balance \leftarrow 0$ 
     $root \leftarrow rotacionSimple(root, \neg dir)$ 
  else
     $ajustarBalance(root, dir, bal)$ 
     $root \leftarrow rotacionDoble(root, \neg dir)$ 
  end if

  res  $\leftarrow root$ 
Complejidad :  $O(1)$ 

```

```

rotacionSimple (in/out  $ab(nodoAvl): root$ , in  $bool: dir$ )  $\rightarrow$  res:  $ab(nodoAvl)$ 
  hijo:  $ab(nodoAvl) \leftarrow subArbol(root, \neg dir)$ 

   $subArbol(root, \neg dir) \leftarrow subArbol(hijo, dir)$ 
   $subArbol(hijo, dir) \leftarrow root$ 

  res  $\leftarrow hijo$ 
Complejidad :  $O(1)$ 

```

```

rotacionDoble (in/out  $ab(nodoAvl): root$ , in  $bool: dir$ )  $\rightarrow$  res:  $ab(nodoAvl)$ 
  nieto:  $ab(nodoAvl) \leftarrow subArbol(subArbol(root, \neg dir), dir)$ 

   $subArbol(subArbol(root, \neg dir), dir) \leftarrow subArbol(nieto, \neg dir)$ 
   $subArbol(nieto, \neg dir) \leftarrow subArbol(root, \neg dir)$ 
   $subArbol(root, \neg dir) \leftarrow nieto$ 

   $nieto \leftarrow subArbol(root, \neg dir)$ 
   $subArbol(root, \neg dir) \leftarrow subArbol(nieto, dir)$ 
   $subArbol(nieto, dir) \leftarrow root$ 

  res  $\leftarrow nieto$ 

```



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

```

ajustarBalance (in/out ab(nodoAvl): root, in bool: dir, in int: bal)
  hijo: ab(nodoAvl) ← subArbol(root, dir)                                O(1)
  nieto: ab(nodoAvl) ← subArbol(hijo, ¬dir)                             O(1)

  if (Raiz(nieto).balance = 0) then                                     O(1)
    Raiz(root).balance ← 0                                           O(1)
    Raiz(hijo).balance ← 0                                           O(1)
  else
    if (Raiz(nieto).balance = bal) then                               O(1)
      Raiz(root).balance ← -bal                                       O(1)
      Raiz(hijo).balance ← 0                                          O(1)
    else
      Raiz(root).balance ← 0                                          O(1)
      Raiz(hijo).balance ← bal                                         O(1)
    end if
  end if

  Raiz(nieto).balance ← 0                                             O(1)
Complejidad :  $O(1)$ 

```

```

iBorrar (in/out diccLog(α): d, in nat: c)
  it: ab(nodoAvl) ← d.abAvl                                           O(1)
  padre: ab(nodoAvl) ← Nil                                           O(1)
  padreDir: bool ← false                                             O(1)
  up: pila(puntero(ab(nodoAvl)))                                       O(1)
  upd: pila(bool)                                                     O(1)

  while (Raiz(it).clave ≠ c)                                           O(1)
    Apilar(upd, (Raiz(it).clave < c))                                  O(1)
    Apilar(up, puntero(it))                                           O(1)

    padre ← it                                                         O(1)
    padreDir ← Tope(upd)                                              O(1)

    it ← subArbol(it, Tope(upd))                                       O(1)
  do                                                                    O(log(#claves(d)))

  if (Raiz(it).clave = Raiz(*d.minimo).clave) then                   O(1)
    if (Nil?(padre)) then                                             O(1)
      d.minimo ← NULL                                                O(1)
    else
      d.minimo ← puntero(padre)                                       O(1)
    end if
  end if

  if (Nil?(Izq(it)) ∨ Nil?(Der(it))) then                             O(1)
    dir: bool ← Nil?(Izq(it))                                         O(1)

    if (Tamano(up) > 1) then                                           O(1)
      SubArbol(*Tope(up), Tope(upd)) ← subArbol(it, dir)           O(1)
    else
      d.abAvl ← subArbol(it, dir)                                     O(1)
    end if

```

```

else
    heredero: ab(nodoAvl) ← Der(it)                                O(1)

    Apilar(Tope(upd), true)                                         O(1)
    Apilar(Tope(up), puntero(it))                                  O(1)

    while(¬Nil?(Izq(heredero))                                     O(1)
        Apilar(upd, false)                                         O(1)
        Apilar(up, puntero(heredero))                             O(1)
        heredero ← Izq(heredero)                                   O(1)
    do                                                                O(log(#claves(d)))

    subArbol(*Tope(up), Tope(up) = puntero(it)) ← Der(heredero)    O(1)

    Izq(heredero) ← Izq(it)                                         O(1)
    Der(heredero) ← Der(it)                                         O(1)

    if(¬Nil?(padre)) then                                           O(1)
        subArbol(padre, padreDir) ← heredero                       O(1)
    end if
end if

break: bool ← false                                               O(1)
while((Tamano(up) > 0) ∧ ¬break)                                    O(1)
    if(Tope(upd)) then                                              O(1)
        Raiz(*Tope(up)).balance ← Raiz(*Tope(up)).balance + 1    O(1)
    else
        Raiz(*Tope(up)).balance ← Raiz(*Tope(up)).balance - 1    O(1)
    end if

    if(abs(Raiz(*Tope(up)).balance) = 1) then                       O(1)
        break ← true                                               O(1)
    else
        if(abs(Raiz(*Tope(up)).balance) > 1) then                 O(1)
            *Tope(up) ← removerBalanceo(*Tope(up), Tope(upd), &break) O(1)
            if(Tamano(up) > 1) then                                  O(1)
                upTope: puntero(ab(nodoAvl)) ← copy(Tope(up))    O(1)
                Desapilar(up)                                       O(1)
                Desapilar(upd)                                       O(1)
                subArbol(*Tope(up), Tope(upd)) ← *upTope          O(1)
            else
                d.abAvl ← *Tope(up)                                  O(1)
            end if
        else
            Desapilar(up)                                           O(1)
            Desapilar(upd)                                           O(1)
        end if
    end if
end if
do                                                                O(log(#claves(d)))
Complejidad :  $O(\log(\#claves(d)) + copy(data))$ 

```

```

removerBalanceo (in/out ab(nodoAvl): root, in bool: dir, in/out puntero(bool): done) → res: ab(nodoAvl)
    hijo: ab(nodoAvl) ← subArbol(root, ¬dir)                        O(1)

    if(dir) then                                                    O(1)
        bal ← 1                                                    O(1)
    else

```

```

    bal ← -1
end if

if (Raiz(hijo).balance = -bal) then
    Raiz(root).balance ← 0
    Raiz(hijo).balance ← 0
    root ← rotacionSimple(root, dir)
else
    if (Raiz(hijo).balance = bal) then
        ajustarBalance(root, ¬dir, -bal)
        root ← rotacionDoble(root, dir)
    else
        Raiz(root).balance ← -bal
        Raiz(hijo).balance ← bal
        root ← rotacionSimple(root, dir)
        *done ← true
    end if
end if

res ← root
Complejidad :  $O(1)$ 

```

```

iMínimo (in  $diccLog(\alpha): d$ ) → res:  $\alpha$ 
    res ← Raiz(*d.minimo).data
Complejidad :  $O(1)$ 

```

```

iDefinido? (in  $diccLog(\alpha): d$ , in  $nat: c$ ) → res: bool
    definido:bool ← false
    it:ab(nodoAvl) ← d.abAvl

    while (¬Nil?(it) ∧ ¬definido) do
        definido ← (Raiz(it).clave = c)
        it ← subArbol(it, (Raiz(it).clave < c))
    end while

    res ← definido
Complejidad :  $O(\log(\#claves(d)))$ 

```

```

iObtener (in/out  $diccLog(\alpha): d$ , in  $nat: c$ ) → res:  $\alpha$ 
    it:ab(nodoAvl) ← d.abAvl

    while (Raiz(it).clave ≠ c) do
        it ← subArbol(it, (Raiz(it).clave < c))
    end while

    res ← Raiz(it).data
Complejidad :  $O(\log(\#claves(d)))$ 

```

```

iVacio? (in  $diccLog(\alpha): d$ ) → res: bool
    res ← Nil?(d.abAvl)
Complejidad :  $O(1)$ 

```

```

inorder (in diccLog( $\alpha$ ): d)  $\rightarrow$  res: lista(tupla(nat,  $\alpha$ ))
  root:ab(nodoAvl)  $\leftarrow$  d.abAvl                                O(1)
  p:pila(puntero(ab(nodoAvl)))  $\leftarrow$  Vacía()                    O(1)
  done:bool  $\leftarrow$  false                                         O(1)
  res  $\leftarrow$  Vacía()                                             O(1)

  while (!done) do                                              O(1)
    if ( $\neg$ Nil?(root)) then                                     O(1)
      Apilar(p, puntero(root))                                O(1)
      root  $\leftarrow$  Izq(root)                                     O(1)
    else
      if  $\neg$ EsVacía?(p) then                                     O(1)
        AgregarAtras(res, <Raiz(*Tope(p)).clave, Raiz(*Tope(p)).data>) O(1)
        root  $\leftarrow$  Der(*Tope(p))                             O(1)
      else
        done  $\leftarrow$  true                                         O(1)
      end if
    end if
  end while
  end while
Complejidad :  $O(\#claves(d))$ 

```

```

• = • (in diccLog( $\alpha$ ): d1, in diccLog( $\alpha$ ): d2)  $\rightarrow$  res: bool
  res  $\leftarrow$  inorder(d1) = inorder(d2)                            O(max(#claves(d1), #claves(d2)))
Complejidad :  $O(\max(\#claves(d1), \#claves(d2)))$ 

```

## 5. Módulo Árbol binario( $\alpha$ )

### 5.1. Interfaz

**se explica con:**  $\text{ÁRBOL BINARIO}(\alpha)$ .

**géneros:**  $\text{ab}(\alpha)$ .

#### 5.1.1. Operaciones básicas de Árbol binario( $\alpha$ )

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

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

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

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

**Descripción:** Crea un árbol binario nulo

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

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

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

**Complejidad:**  $O(\text{copy}(r) + \text{copy}(i) + \text{copy}(d))$

**Descripción:** Crea un árbol binario con hijo izquierdo  $i$ , hijo derecho  $d$  y raíz de valor  $r$

$\text{RAÍZ}(\text{in/out } a : \text{ab}(\alpha)) \rightarrow res : \alpha$

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

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{raíz}(a))\}$

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

**Descripción:** Devuelve el valor de la raíz del árbol

**Aliasing:**  $res$  es modificable si y sólo si  $a$  lo es

$\text{IZQ}(\text{in/out } a : \text{ab}(\alpha)) \rightarrow res : \text{ab}(\alpha)$

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

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{izq}(a))\}$

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

**Descripción:** Devuelve el hijo izquierdo

**Aliasing:**  $res$  es modificable si y sólo si  $a$  lo es

$\text{DER}(\text{in/out } a : \text{ab}(\alpha)) \rightarrow res : \text{ab}(\alpha)$

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

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{der}(a))\}$

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

**Descripción:** Devuelve el hijo derecho

**Aliasing:**  $res$  es modificable si y sólo si  $a$  lo es

$\text{NIL?}(\text{in/out } a : \text{ab}(\alpha)) \rightarrow res : \text{bool}$

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

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

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

**Descripción:** Devuelve **true** si  $res$  es un árbol vacío

## 5.2. Representación

### 5.2.1. Representación de $\text{ab}(\alpha)$

$\text{ab}(\alpha)$  se representa con *estr*

donde *estr* es puntero(*nodoAb*)

donde *nodoAb* es tupla( *raiz*:  $\alpha$ , *hijos*: arreglo[2] de  $\text{ab}(\alpha)$  )

### 5.2.2. Invariante de Representación

- (I) No puede haber ciclos en el árbol
- (II) Los hijos no pueden apuntar a un mismo árbol

### 5.2.3. Función de Abstracción

$\text{Abs} : \text{estr } e \longrightarrow \text{ab}(\alpha) \quad \{\text{Rep}(e)\}$   
 $\text{Abs}(e) =_{\text{obs}} \text{abn} : \text{ab}(\alpha) \mid (\text{nil?}(\text{abn}) \Leftrightarrow e = \text{NULL}) \wedge$   
 $\quad (\neg \text{nil?}(\text{abn}) \Rightarrow_{\text{L}} (\text{raíz}(\text{abn}) = e \rightarrow \text{raíz} \wedge \text{izq}(\text{abn}) = e \rightarrow \text{hijos}[0] \wedge \text{der}(\text{abn}) = e \rightarrow \text{hijos}[1]))$

## 5.3. Algoritmos

iNil ()  $\rightarrow$  res:  $\text{ab}(\alpha)$

res  $\leftarrow$  NULL

O(1)

**Complejidad :** O(1)

iBin (**in**  $i : \text{ab}(\alpha)$ , **in**  $r : \alpha$ , **in**  $d : \text{ab}(\alpha)$ )  $\rightarrow$  res:  $\text{ab}(\alpha)$

nuevoAb:nodoAb

O(1)

nuevoAb.raiz  $\leftarrow$  copy(r)

O(copy(r))

nuevoAb.hijos[0]  $\leftarrow$  copy(i)

O(copy(i))

nuevoAb.hijos[1]  $\leftarrow$  copy(d)

O(copy(d))

res  $\leftarrow$  puntero(nuevoAb)

O(1)

**Complejidad :** O(copy(r) + copy(i) + copy(d))

iRaíz (**in/out**  $a : \text{ab}(\alpha)$ )  $\rightarrow$  res:  $\alpha$

res  $\leftarrow$  ( $a \rightarrow \text{raíz}$ )

O(1)

**Complejidad :** O(1)

iIzq (**in/out**  $a : \text{ab}(\alpha)$ )  $\rightarrow$  res:  $\text{ab}(\alpha)$

$\text{res} \leftarrow (a \rightarrow \text{hijos}[0])$

$O(1)$

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

iDer (**in/out**  $a : \text{ab}(\alpha)$ )  $\rightarrow$  res:  $\text{ab}(\alpha)$

$\text{res} \leftarrow (a \rightarrow \text{hijos}[1])$

$O(1)$

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

iNil? (**in**  $a : \text{ab}(\alpha)$ )  $\rightarrow$  res: bool

$\text{res} \leftarrow (a = \text{NULL})$

$O(1)$

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

## 6. Módulo Diccionario $\text{String}(\alpha)$

### 6.1. Interfaz

se explica con:  $\text{DICCIONARIO}(\text{STRING}, \alpha)$ . **géneros:**  $\text{diccString}(\alpha)$ .

Se representa mediante un árbol n-ario con invariante de trie

$\text{CREARDICC}() \rightarrow res : \text{diccString}(\alpha)$

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

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

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

**Descripción:** Crea un diccionario vacío.

$\text{DEFINIDO?}(\text{in } d : \text{diccString}(\alpha), \text{in } c : \text{string}) \rightarrow res : \text{bool}$

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

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

**Complejidad:**  $O(L)$

**Descripción:** Devuelve true si la clave está definida en el diccionario y false en caso contrario.

$\text{DEFINIR}(\text{in } d : \text{diccString}(\alpha), \text{in } c : \text{string}, \text{in } s : \alpha)$

**Pre**  $\equiv \{d =_{\text{obs}} d_0\}$

**Post**  $\equiv \{d =_{\text{obs}} \text{definir}(c, s, d_0)\}$

**Complejidad:**  $O(L)$

**Descripción:** Define la clave  $c$  con el significado  $s$

**Aliasing:** Almacena una copia de  $s$ .

$\text{OBTENER}(\text{in } d : \text{diccString}(\alpha), \text{in } c : \text{string}) \rightarrow res : \alpha$

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

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

**Complejidad:**  $O(L)$

**Descripción:** Devuelve el significado correspondiente a la clave  $c$ .

**Aliasing:** Devuelve el significado almacenado en el diccionario, por lo que  $res$  es modificable si y sólo si  $d$  lo es.

$\bullet = \bullet(\text{in/out } d : \text{diccString}(\alpha), \text{in/out } d' : \text{diccString}(\alpha)) \rightarrow res : \text{bool}$

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

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

**Complejidad:**  $O(L * n * (\alpha =_{\text{obs}} \alpha'))$

**Descripción:** Indica si  $d$  es igual  $d'$

$\text{COPIAR}(\text{in } \text{dicc} : \text{diccString}(\alpha)) \rightarrow res : \text{diccString}(\alpha)$

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

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

**Complejidad:**  $O(n * L * \text{copy}(\alpha))$

**Descripción:** Devuelve una copia del diccionario