

# 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>3</b>
1.1. Interfaz	3
1.1.1. Operaciones básicas de DCNet	3
1.2. Representación	4
1.2.1. Representación de dcnet	4
1.2.2. Invariante de Representación	4
1.2.3. Función de Abstracción	7
<b>2. Módulo Red</b>	<b>9</b>
2.1. Interfaz	9
2.2. Representación	10
2.2.1. Estructura	10
2.2.2. Invariante de Representación	10
2.2.3. Función de Abstracción	13
<b>3. Módulo Cola de mínima prioridad(<math>\alpha</math>)</b>	<b>14</b>
3.1. Especificación	14
3.2. Interfaz	15
3.2.1. Operaciones básicas de Cola de mínima prioridad	15
3.3. Representación	15
3.3.1. Representación de colaMinPrior	15
3.3.2. Invariante de Representación	15
3.3.3. Función de Abstracción	16
3.4. Algoritmos	16
<b>4. Módulo Diccionario AVL(<math>\alpha</math>)</b>	<b>17</b>
4.1. Interfaz	17
4.1.1. Operaciones básicas de Diccionario AVL( $\alpha$ )	17
4.1.2. Operaciones auxiliares del TAD	17
4.2. Representación	18
4.2.1. Representación de $\text{dicc}_{avl}(\alpha)$	18
4.2.2. Invariante de Representación	18
4.2.3. Función de Abstracción	18
4.3. Algoritmos	19
<b>5. Módulo Trie(<math>\alpha</math>)</b>	<b>21</b>
5.1. Interfaz	21

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

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

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

**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*: dcnet, **in** *c*: compu)  $\rightarrow$  *res* : conj(paquete)

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

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

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

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

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

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

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

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

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

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

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

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

## 1.2. Representación

### 1.2.1. Representación de dcnet

dcnet se representa con estr

donde estr es tupla(*topología*: red,  
                           *vectorCompusDCNet*: vector(compuDCNet),  
                           *diccCompusDCNet*: dicc<sub>trie</sub>(puntero(compuDCNet)),  
                           *conjPaquetesDCNet*: conj(paqueteDCNet),  
                           *laQueMásEnvio*: puntero(compuDCNet))

donde compuDCNet es tupla(*pc*: puntero(compu),  
                               *conjPaquetes*: conj(paquete),  
                               *diccPaquetesDCNet*: dicc<sub>avl</sub>(nat, 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) `laQueMásEnvio` es un puntero a la `compuDCNet` en `vectorCompusDCNet` que más paquetes enviados tiene. Si no hay compus es `NULL`
- (V) Todos los paquetes en `conjPaquetes` de cada `compuDCNet` tienen id único y tanto su origen como destino existen en la topología
- (VI) 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
- (VII) Las claves de `diccPaquetesDCNet` son los id de los paquetes en `conjPaquetes`
- (VIII) Los significados de `diccPaquetesDCNet` contienen un `itConj` que apunta al paquete con el id equivalente a su clave y en recorrido, un camino mínimo válido para el origen del paquete y la compu en la que se encuentra
- (IX) La `colaPaquetesDCNet` es vacía si y sólo si `conjPaquetes` lo es, si no lo es, su próximo es un puntero a un paqueteDCNet de `diccPaquetesDCNet` que contiene un `itConj` cuyo siguiente es uno de los paquetes de `conjPaquetes` con mayor prioridad
- (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

Rep : `estr`  $\longrightarrow$  `bool`

$$\begin{aligned}
\text{Rep}(e) \equiv & \text{true} \iff \\
& (\#(\text{computadoras}(e.\text{topologia})) = \text{long}(e.\text{vectorCompusDCNet}) = \#(\text{claves}(e.\text{diccCompusDCNet}))) \wedge_L \\
& (\forall c: \text{compu})(c \in \text{computadoras}(e.\text{topologia}) \Rightarrow \\
& \quad ( \\
& \quad (\exists cd: \text{compuDCNet}) (\text{está?}(cd, e.\text{vectorCompusDCNet}) \wedge cd.\text{pc} = \text{puntero}(c)) \wedge \\
& \quad (\exists s: \text{string})(\text{def?}(s, e.\text{diccCompusDCNet}) \wedge s = c.\text{ip}) \\
& \quad ) \\
& ) \wedge_L \\
& (\forall cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \Rightarrow_L \\
& \quad (\exists s: \text{string}) (\text{def?}(s, e.\text{diccCompusDCNet}) \wedge \\
& \quad s = cd.\text{pc} \rightarrow \text{ip} \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.\text{pc}) = \text{compuQueMásEnvió}(e.\text{vectorCompusDCNet}) \wedge e.\text{laQueMásEnvió} = \text{puntero}(cd)) \wedge_L \\
& (\forall cd_1: \text{compuDCNet})(\text{está?}(cd_1, e.\text{vectorCompusDCNet}) \Rightarrow \\
& \quad (\forall p_1: \text{paquete})(p_1 \in cd_1.\text{conjPaquetes} \Rightarrow \\
& \quad (\forall cd_2: \text{compuDCNet})(\text{está?}(cd_2, e.\text{vectorCompusDCNet}) \wedge cd_1 \neq cd_2) \Rightarrow \\
& \quad (\forall p_2: \text{paquete})(p_2 \in cd_2.\text{conjPaquetes} \Rightarrow p_1.\text{id} \neq p_2.\text{id}) \\
& \quad ) \\
& ) \\
& ) \wedge_L \\
& (\forall cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \Rightarrow \\
& \quad ( \\
& \quad (\#(cd.\text{conjPaquetes}) = \#(\text{claves}(cd.\text{diccPaquetesDCNet}))) \wedge_L \\
& \quad (\forall p: \text{paquete})(p \in cd.\text{conjPaquetes} \Rightarrow \\
& \quad ( \\
& \quad ((p.\text{origen} \in \text{computadoras}(e.\text{topologia}) \wedge p.\text{destino} \in \text{computadoras}(e.\text{topologia}) \wedge \\
& \quad p.\text{destino} \neq *(cd.\text{pc})) \wedge_L \\
& \quad (\exists sc: \text{secu}(\text{compu}))(sc \in \text{caminosMinimos}(e.\text{topologia}, p.\text{origen}, p.\text{destino}) \wedge \text{está?}(*(cd.\text{pc}), sc))) \wedge \\
& \quad (\exists n: \text{nat}) ((\text{def?}(n, cd.\text{diccPaquetesDCNet}) \wedge p.\text{id} = n) \wedge_L \\
& \quad (\text{Siguiente}(\text{obtener}(n, e.\text{diccPaquetesDCNet}).\text{it}) = p \wedge \\
& \quad ((p.\text{origen} = *(cd.\text{pc}) \wedge \text{obtener}(n, e.\text{diccPaquetesDCNet}).\text{recorrido} = *(cd.\text{pc}) \bullet \langle \rangle) \vee \\
& \quad (p.\text{origen} \neq *(cd.\text{pc}) \wedge \\
& \quad \text{obtener}(n, e.\text{diccPaquetesDCNet}).\text{recorrido} \in \text{caminosMinimos}(e.\text{topologia}, p.\text{origen}, *(cd.\text{pc})))) \\
& \quad ) \\
& ) \wedge_L \\
& (\emptyset?(cd.\text{conjPaquetes}) \Leftrightarrow \text{vacía?}(cd.\text{colaPaquetesDCNet})) \wedge \\
& (\neg \text{vacía?}(cd.\text{colaPaquetesDCNet}) \Rightarrow_L \\
& \quad (\exists n: \text{nat})(\text{def?}(n, cd.\text{diccPaquetesDCNet}) \wedge_L \\
& \quad ( \\
& \quad \text{Siguiente}(\text{obtener}(n, cd.\text{diccPaquetesDCNet}).\text{it}) = \text{paqueteMásPrioridad}(cd.\text{conjPaquetes}) \wedge \\
& \quad \text{proximo}(cd.\text{colaPaquetesDCNet}) = \text{puntero}(\text{obtener}(n, cd.\text{diccPaquetesDCNet})) \\
& \quad )) \\
& ) \wedge \\
& (cd.\text{enviados} \geq \text{enviadosCompu}(*(cd.\text{pc}), e.\text{vectorCompusDCNet})) \\
& ) \\
& )
\end{aligned}$$

$\text{compuQueMásEnvió} : \text{secu}(\text{compuDCNet}) \text{ } scd \longrightarrow \text{compu}$	$\{\neg \text{vacía?}(scd)\}$
$\text{maxEnviado} : \text{secu}(\text{compuDCNet}) \text{ } scd \longrightarrow \text{nat}$	$\{\neg \text{vacía?}(scd)\}$
$\text{enviaronK} : \text{secu}(\text{compuDCNet}) \times \text{nat} \longrightarrow \text{conj}(\text{compu})$	
$\text{paqueteMásPrioridad} : \text{conj}(\text{paquete}) \text{ } cp \longrightarrow \text{paquete}$	$\{\neg \emptyset?(cp)\}$
$\text{paquetesConPrioridadK} : \text{conj}(cp) \times \text{nat} \longrightarrow \text{conj}(\text{paquete})$	
$\text{altaPrioridad} : \text{conj}(\text{paquetes}) \text{ } cp \longrightarrow \text{nat}$	$\{\neg \emptyset?(cp)\}$
$\text{enviadosCompu} : \text{compu} \times \text{secu}(\text{compuDCNet}) \longrightarrow \text{nat}$	

```

aparicionesCompu : compu × conj(nat) cn × dicc(nat × paqueteDCNet) dp → nat           {claves(dp) ⊆ cn}

compuQueMásEnvió(scd) ≡ dameUno(environK(scd, maxEnviado(scd)))
maxEnviado(scd) ≡ if vacía?(fin(scd)) then prim(scd).enviados else max(prim(scd), maxEnviado(fin(scd))) fi
environK(scd, k) ≡ if vacía?(scd) then
  ∅
else
  if prim(scd).enviados = k then
    Ag(*(prim(scd).pc), environK(fin(scd), k))
  else
    environK(fin(scd), k)
  fi
fi

paqueteMásPrioridad(dcn, cp) ≡ dameUno(paquetesConPrioridadK(cp, altaPrioridad(cp)))
altaPrioridad(cp) ≡ if ∅?(sinUno(cp)) then
  dameUno(cp).prioridad
else
  min(dameUno(cp).prioridad, altaPrioridad(sinUno(cp)))
fi

paquetesConPrioridadK(cp, k) ≡ if ∅?(cp) then
  ∅
else
  if dameUno(cp).prioridad = k then
    Ag(dameUno(cp), paquetesConPrioridadK(sinUno(cp), k))
  else
    paquetesConPrioridadK(sinUno(cp), k)
  fi
fi

enviadosCompu(c, scd) ≡ 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, dpd) ≡ if ∅?(cn) then
  0
else
  if está?(c, significado(dameUno(cn), dpd).recorrido) then
    1 + aparicionesCompu(c, sinUno(cn), dpd)
  else
    aparicionesCompu(c, sinUno(cn), dpd)
  fi
fi

```

### 1.2.3. Función de Abstracción

Abs : estr  $e$  → dcnnet {Rep( $e$ )}

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

)  
)



## 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:**  $O(L + n)$   
**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:**  $O(L)?$   
**Descripción:** Conecta dos computadoras

**COMPUTADORAS(in r : red)  $\rightarrow res : conj(compu)$**   
**Pre**  $\equiv \{true\}$   
**Post**  $\equiv \{alias(res =_{obs} computadoras(r))\}$   
**Complejidad:**  $O(1)$   
**Descripción:** Devuelve las computadoras de la red [Devuelve una referencia no 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:**  $O(n)$   
**Descripción:** Devuelve el conjunto de computadoras conectadas con c  
**Aliasing:** Devuelve una copia de las computadoras conectadas a c

$$\text{USAINTERFAZ?}(\text{in } r : \text{red}, \text{in } c : \text{compu}, \text{in } i : \text{interfaz}) \rightarrow res : \text{bool}$$
$$\mathbf{Pre} \equiv \{c \in \text{computadoras}(r)\}$$
$$\mathbf{Post} \equiv \{res =_{\text{obs}} \text{usaInterfaz?}(r, c, i)\}$$

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

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

$$\text{CAMINOSMINIMOS}(\text{in } r : \text{red}, \text{in } c : \text{compu}, \text{in } c' : \text{compu}) \rightarrow res : \text{conj}(\text{lista}(\text{compu}))$$
$$\mathbf{Pre} \equiv \{(c \in \text{computadoras}(r)) \wedge (c' \in \text{computadoras}(r))\}$$
$$\mathbf{Post} \equiv \{\text{alias}(res =_{\text{obs}} \text{caminosMinimos}(r, c, i))\}$$

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

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

**Aliasing:** Devuelve una referencia no modificable

$$\text{HAYCAMINO?}(\text{in } r : \text{red}, \text{in } c : \text{compu}, \text{in } c' : \text{compu}) \rightarrow \text{res} : \text{bool}$$
$$\mathbf{Pre} \equiv \{(c \in \text{computadoras}(r)) \wedge (c' \in \text{computadoras}(r))\}$$
$$\mathbf{Post} \equiv \{res =_{\text{obs}} \text{hayCamino?}(r, c, i)\}$$

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

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

$$\text{COPIAR}(\mathbf{in} \ r : \mathbf{red}) \rightarrow res : \mathbf{red}$$
$$\mathbf{Pre} \equiv \{\text{true}\}$$
$$\mathbf{Post} \equiv \{res =_{\text{obs}} r\}$$

**Complejidad:**  $O(?)$

**Descripción:** Devuelve una copia la red

$$\bullet = \bullet(\text{in } r : \text{red}, \text{in } r' : \text{red}) \rightarrow \text{res} : \text{bool}$$
$$\mathbf{Pre} \equiv \{\text{true}\}$$
$$\mathbf{Post} \equiv \{res =_{\text{obs}} (r =_{\text{obs}} r')\}$$

**Complejidad:**  $O(?)$

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

## 2.2. Representación

### 2.2.1. Estructura

red se representa con estr

donde **estr** es  $\text{tupla}(\text{compus: conj}(\text{compu}) ,$   
 $\text{dns: dicc}_{Trie}(\text{nodoRed}) )$

donde `nodoRed` es  $\text{tupla}(\text{pc: puntero}(\text{compu}) ,$   
 $\text{caminos: dicc}_{Trie}(\text{conj}(\text{lista}(\text{compu}))) ,$   
 $\text{conexiones: dicc}_{Lineal}(\text{nat}, \text{puntero}(\text{nodoRed}))$  )

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

### 2.2.2. Invariante de Representación

- (I) Todos los elementos de *compus* deben tener IPs distintas.

- (II) Para cada compu, el trie *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.compuls**), 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  $\iff$  (

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

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

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

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

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

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

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

)

vecinas	: nodoRed	$\rightarrow$ conj(nodoRed)
auxVecinas	: nodoRed $\times$ dicc(nat $\times$ puntero(nodoRed))	$\rightarrow$ conj(nodoRed)
secusDeLongK	: conj(secu( $\alpha$ )) $\times$ nat	$\rightarrow$ conj(secu( $\alpha$ ))
longMenorSec	: conj(secu( $\alpha$ )) <i>secus</i>	$\rightarrow$ nat $\{ \neg \emptyset?(\text{secus}) \}$
darRutas	: nodoRed $nA \times$ nodoRed $nB \times$ conj(pc) $\times$ secu(nodoRed)	$\rightarrow$ conj(secu(nodoRed))
darRutasVecinas	: conj(pc) <i>vec</i> $\times$ nodoRed $n \times$ conj(pc) $\times$ secu(nodoRed)	$\rightarrow$ conj(secu(nodoRed))
darCaminosMinimos	: nodoRed $n1 \times$ nodoRed $n1$	$\rightarrow$ conj(secu(compu))

vecinas( $n$ )  $\equiv$  auxVecinas( $n, n.conexiones$ )

auxVecinas( $n, cs$ )  $\equiv$  **if**  $\emptyset?(cs)$  **then**  
 $\emptyset$   
**else**  
 Ag( $\text{obtener}(\text{dameUno}(\text{claves}(cs)), cs)$ , auxVecinas( $n, \text{sinUno}(cs)$ ))  
**fi**

```

secusDeLongK(secus, k)           ≡ if  $\emptyset?(secus)$  then
                                    $\emptyset$ 
                                   else
                                   if long(dameUno(secus)) = k then
                                   dameUno(secus)  $\cup$  secusDeLongK(sinUno(secus), k)
                                   else
                                   secusDeLongK(sinUno(secus), k)
                                   fi
                                   fi
longMenorSec(secus)             ≡ if  $\emptyset?(sinUno(secus))$  then
                                   long(dameUno(secus))
                                   else
                                   min(long(dameUno(secus)),
                                   longMenorSec(sinUno(secus)))
                                   fi
darRutas(nA, nB, rec, ruta)    ≡ if nB  $\in$  vecinas(nA) then
                                   Ag(ruta  $\circ$  nB,  $\emptyset$ )
                                   else
                                   if  $\emptyset?(vecinas(nA) - rec)$  then
                                    $\emptyset$ 
                                   else
                                   darRutas(dameUno(vecinas(nA) - rec),
                                   nB, Ag(nA, rec),
                                   ruta  $\circ$  dameUno(vecinas(nA) - rec))  $\cup$ 
                                   darRutasVecinas(sinUno(vecinas(nA) - rec),
                                   nB, Ag(nA, rec),
                                   ruta  $\circ$  dameUno(vecinas(nA) - rec))
                                   fi
                                   fi
darRutasVecinas(vecinas, n, rec, ruta) ≡ if  $\emptyset?(vecinas)$  then
                                    $\emptyset$ 
                                   else
                                   darRutas(dameUno(vecinas), n, rec, ruta)  $\cup$ 
                                   darRutasVecinas(sinUno(vecinas), n, rec, ruta)
                                   fi
darCaminosMinimos(nA, nB)      ≡ secusDeLongK(darRutas(nA, nB,  $\emptyset$ ,  $\langle > \rangle$ ),
                                   longMenorSec(darRutas(nA, nB,  $\emptyset$ ,  $\langle > \rangle$ )))

```

### 2.2.3. Función de Abstracción

Abs : estr *e*  $\longrightarrow$  red {Rep(*e*)}

Abs(*e*) =<sub>obs</sub> r: red | *e*.compus =<sub>obs</sub> computadoras(*r*)  $\wedge$   
 $((\forall c1, c2: compu, i1, i2: string, n1, n2: nodoRed) ($   
 $(c1 \in e.compus \wedge i1 = c1.ip \wedge def?(i1, e.dns) \wedge_L n1 = obtener(i1, e.dns) \wedge c1 = *n1.pc) \wedge$   
 $(c2 \in e.compus \wedge i2 = c2.ip \wedge def?(i2, e.dns) \wedge_L n2 = obtener(i2, e.dns) \wedge c2 = *n2.pc) \wedge$   
 $(c1 \neq c2)) ) \Rightarrow_L$   
 $(conectadas?(r, c1, c2) \Leftrightarrow ( (\exists t1, t2: nat) ($   
 $t1 = interfazUsada(r, c1, c2) \wedge t2 = interfazUsada(r, c2, c1) \wedge$   
 $def?(t1, n1.conexiones) \wedge def?(t2, n2.conexiones) \wedge_L ($   
 $\&n2 = obtener(t1, n1.conexiones) \wedge \&n1 = obtener(t2, n2.conexiones)$   
 $))))$

### 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>

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

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

**usa** **BOOL**

**observadores básicos**

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

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

$\{\neg \text{vacía?}(c)\}$

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

$\{\neg \text{vacía?}(c)\}$

**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

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

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

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

**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

### 3.3. Representación

#### 3.3.1. Representación de colaMinPrior

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

donde **estr** es  $\text{dicc}_{\text{avl}}(\text{nat}, \text{nodoEncolados})$

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

#### 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

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

$$\begin{aligned} \text{Rep}(e) &\equiv \text{true} \iff \\ &(\forall n : \text{nat}) \text{ def?}(n, e) \Rightarrow_{\text{L}} ((\text{obtener}(n, e).\text{prioridad} = n) \wedge \\ &\neg \text{vacía?}(\text{obtener}(n, e).\text{encolados})) \end{aligned}$$

### 3.3.3. Función de Abstracción

$$\begin{aligned} \text{Abs} : \text{estr } e &\longrightarrow \text{colaMinPrior} && \{\text{Rep}(e)\} \\ \text{Abs}(e) =_{\text{obs}} \text{cmp} : \text{colaMinPrior} &| \text{ (vacía?}(\text{cmp}) \Leftrightarrow (\# \text{claves}(e) = 0)) \wedge \\ &\neg \text{vacía?}(\text{cmp}) \Rightarrow_{\text{L}} \\ &((\text{próximo}(\text{cmp}) = \text{próximo}(\text{mínimo}(e).\text{encolados})) \wedge \\ &(\text{desencolar}(\text{cmp}) = \text{desencolar}(\text{mínimo}(e).\text{encolados}))) \end{aligned}$$

## 3.4. Algoritmos

```
iVacía () → res: colaMinPrior(α)

res ← Vacío ()

Complejidad : O(1)
```

```
iVacía? (in c: colaMinPrior(α)) → res: bool

res ← (#Claves(c) = 0)

Complejidad : O(1)
```

```
iDesencolar (in/out c: colaMinPrior(α)) → res: α

res ← Copiar(Proximo(Minimo(c).encolados))
Desencolar(Minimo(c).encolados)
if EsVacía?(Minimo(c).encolados) then
  Borrar(c, Minimo(c).prioridad)
end if

Complejidad : O(log(tamaño(c)) + O(copy(α)))
```

```
iEncolar (in/out c: colaMinPrior(α), in p: nat, in a: α)

if Definido?(p) then
  Encolar(Significado(c, p).encolados, a)
else
  nodoEncolados nuevoNodoEncolados
  nuevoNodoEncolados.encolados ← Vacía()
  nuevoNodoEncolados.prioridad ← p
  Encolar(nuevoNodoEncolados.encolados, a)
  Definir(c, p, nuevoNodoEncolados)
end if

Complejidad : O(log(tamaño(c)) + O(copy(α)))
```



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

### 4.1. Interfaz

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

géneros:  $\text{dicc}_{avl}(\alpha)$ .

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

$\text{CREARDICC}() \rightarrow res : \text{dicc}_{avl}(\alpha)$

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

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

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

**Descripción:** Crea un diccionario vacío

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

**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

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

**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  $d$

$\text{OBTENER}(\text{in } c : \text{string}, \text{in/out } d : \text{dicc}_{avl}(\alpha)) \rightarrow res : \alpha$

**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

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

**Pre**  $\equiv \{\#claves(d) > 0\}$

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

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

**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

#### 4.1.2. Operaciones auxiliares del TAD

$\text{claveMínima} : \text{dicc}(\text{nat} \times \alpha) \ d \longrightarrow \text{nat} \quad \{\#claves(d) > 0\}$

$\text{darClaveMínima} : \text{dicc}(\text{nat} \times \alpha) \ d \times \text{conj}(\text{nat}) \ c \longrightarrow \text{nat} \quad \{(\#claves(d) > 0) \wedge (c \subseteq \text{claves}(d))\}$

$\text{claveMínima}(d) \equiv \text{darClaveMínima}(d, \text{claves}(d))$

$\text{darClaveMínima}(d, c) \equiv \text{if } \emptyset?(\text{sinUno}(c)) \text{ then}$   
      $\text{dameUno}(c)$   
   **else**  
      $\text{min}(\text{dameUno}(c), \text{darClaveMínima}(d, \text{sinUno}(c)))$   
   **fi**

## 4.2. Representación

### 4.2.1. Representación de $\text{dicc}_{avl}(\alpha)$

$\text{dicc}_{avl}(\alpha)$  se representa con **estr**

donde **estr** es puntero(nodoAvl)

donde **nodoAvl** es tupla( *clave*: nat, *data*:  $\alpha$ , *balance*: int, *hijos*: arreglo[2] de puntero(nodoAvl) )

### 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 existe un nodo en el último nivel de balanceo tal que tenga un hijo a la izq
  - \* -1 si existe un nodo en el último nivel de balanceo tal que tenga un hijo a la der
- (III) **estr** está balanceado entero o hasta ( $\text{altura}(\text{estr}) - 1$ ) está balanceado
- (IV) Todas las claves son distintas.

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

$\text{Rep}(e) \equiv \text{true} \iff \text{esABB}(e) \wedge \text{balanceadoBien}(e) \wedge \text{clavesDistintas}(e, \text{vacío})$

$\text{esABB} : \text{puntero}(\text{nodoAvl}) \rightarrow \text{bool}$

$\text{alturaBien} : \text{puntero}(\text{nodoAvl}) \rightarrow \text{bool}$

$\text{balanceadoBien} : \text{puntero}(\text{nodoAvl}) \times \text{conj}(\text{nat}) \rightarrow \text{bool}$

$\text{esABB}(n) \equiv (n \neq \text{NULL}) \Rightarrow_L ($   
 $((\text{prim}(n.\text{hijos}) \neq \text{NULL}) \Rightarrow_L (n.\text{clave} > \text{prim}(n.\text{hijos}) \wedge \text{esABB}(\text{prim}(n.\text{hijos})))) \wedge$   
 $((\text{ult}(n.\text{hijos}) \neq \text{NULL}) \Rightarrow_L (n.\text{clave} < \text{ult}(n.\text{hijos}) \wedge \text{esABB}(\text{ult}(n.\text{hijos}))))))$

$\text{balanceadoBien}(n) \equiv (\text{balanceado}(n) \vee \text{balanceado}(\text{hastaAltura}(n, \text{altura}(n)-1))) \wedge_L (n \neq \text{NULL}) \Rightarrow_L ($   
 $\text{if } ((\text{prim}(n.\text{hijos}) \neq \text{NULL}) \wedge (\text{ult}(n.\text{hijos}) \neq \text{NULL})) \text{ then}$   
 $\quad \text{balanceadoBien}(\text{prim}(n.\text{hijos})) \wedge \text{balanceadoBien}(\text{ult}(n.\text{hijos}))$   
 $\text{else}$   
 $\quad \text{if } (\text{prim}(n.\text{hijos}) \neq \text{NULL}) \text{ then}$   
 $\quad \quad n.\text{balance} = 1$   
 $\quad \text{else}$   
 $\quad \quad \text{if } (\text{prim}(n.\text{hijos}) \neq \text{NULL}) \text{ then } n.\text{balance} = -1 \text{ else } n.\text{balance} = 0 \text{ fi}$   
 $\quad \text{fi } c$   
 $\text{fi } ) \wedge$

$\text{clavesDistintas}(n, cs) \equiv (n \neq \text{NULL}) \Rightarrow_L$   
 $n.\text{clave} \notin cs \wedge$   
 $\text{clavesDistintas}(\text{prim}(n.\text{hijos}), \text{Ag}(n.\text{clave}, cs)) \wedge$   
 $\text{clavesDistintas}(\text{ult}(n.\text{hijos}), \text{Ag}(n.\text{clave}, cs))$

### 4.2.3. Función de Abstracción

$\text{Abs} : \text{estr } e \rightarrow \text{dicc}(\text{nat}, \alpha)$

$\{\text{Rep}(e)\}$

$\text{Abs}(e) =_{\text{obs}} d : \text{dicc}(\text{nat}, \alpha) \mid \text{auxAbs}(e, \text{vacío})$

$\text{auxAbs} : \text{puntero}(\text{nodoAvl}) \times \text{dicc}(\text{nat}; \alpha) \rightarrow \text{dicc}(\text{nat}, \alpha)$

```

auxAbs( $n, d$ )  $\equiv$  if  $n = \text{NULL}$  then
     $d$ 
else
    definir( $n.clave, n.data, \text{auxAbs}(\text{prim}(n.hijos), \text{auxAbs}(\text{ult}(n.hijos), d))$ )
fi

```

### 4.3. Algoritmos

```

iInsertar (in/out puntero(nodoAvl): tree, in nat:  $c$ , in  $\alpha$ :  $s$ )
    if ( $\text{tree} \rightarrow \text{root} = \text{NULL}$ ) then
    {
         $\text{tree} \rightarrow \text{root} = \text{make\_node}(\text{data});$ 

        if ( $\text{tree} \rightarrow \text{root} = \text{NULL}$ )
        {
            return 0;
        }
    }

```

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

```

iVacío ()  $\rightarrow$  res: puntero(nodoAvl)
    res  $\leftarrow$  NULL

```

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

$O(1)$

```

iDefinir (in/out  $p$ : puntero(nodoAvl), in/out clave: nat, in/out significado: alpha)
    padre: puntero(nodoAvl)  $\leftarrow$   $p$   $O(1)$ 
    nuevo: nodoAvl  $\leftarrow$  {clave:  $clave$ , data:  $\text{significado}$ , altura: 0, hijos: [NULL, NULL]}  $O(\text{copy}(\text{significado}))$ 
    bool: buscarMas  $\leftarrow$  true  $O(1)$ 
    dir: int

    while (buscarMas) do  $O(1)$ 
        if ( $\text{padre.clave} = \text{clave}$ ) then  $O(1)$ 
            padre.data  $\leftarrow$   $\text{significado}$   $O(\text{copy}(\text{significado}))$ 
            buscarMas  $\leftarrow$  false  $O(1)$ 
        else
            if ( $\text{padre.clave} > \text{clave}$ ) then  $O(1)$ 
                dir  $\leftarrow$  0  $O(1)$ 
            else
                dir  $\leftarrow$  1  $O(1)$ 
            end if

            if ( $\text{padre.hijos}[\text{dir}] = \text{NULL}$ ) then  $O(1)$ 
                padre.hijos[dir]  $\leftarrow$  &nuevo  $O(1)$ 
                balancear( $p$ )
                buscarMas  $\leftarrow$  false  $O(1)$ 
            else
                padre  $\leftarrow$  padre.hijos[dir]  $O(1)$ 
            end if

        end if
    end while  $O(p.altura)$ 
Complejidad :  $O(p.altura + \text{copy}(\text{significado}))$ 

```

```
iBorrar () → res: puntero(nodoAvl)
  res ← NULL
Complejidad :  $O(1)$ 
```

 $O(1)$ 

```
iDefinido? () → res: puntero(nodoAvl)
  res ← NULL
Complejidad :  $O(1)$ 
```

 $O(1)$ 

```
iObtener () → res: puntero(nodoAvl)
  res ← NULL
Complejidad :  $O(1)$ 
```

 $O(1)$ 

```
iClaves () → res: puntero(nodoAvl)
  res ← NULL
Complejidad :  $O(1)$ 
```

 $O(1)$

## 5. Módulo Trie( $\alpha$ )

### 5.1. Interfaz

se explica con: `DICCIONARIO(String,  $\alpha$ )`. **géneros:** `diccTrie( $\alpha$ )`.

`CREARDICC()`  $\rightarrow res : \text{dicc}_{Trie}(\alpha)$

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

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

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

**Descripción:** Crea un diccionario vacío.

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

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

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

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

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

`DEFINIR(in  $c$ : string, in  $s$ :  $\alpha$ , in/out  $d$ : diccTrie( $\alpha$ ))`

**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  $s$ .

`OBTENER(in  $c$ : string, in  $d$ : diccTrie( $\alpha$ ))`  $\rightarrow res : \alpha$

**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.