# 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

# 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** $\equiv$ {$res =_{\text{obs}}$ 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** $\equiv$ {
$dcn =_{\text{obs}} dcn_0 \wedge$
$\neg($ $(\exists\ p'$:paquete)( paqueteEnTransito($dcn$, $p'$) $\wedge$ id($p$) = id($p'$) $\wedge$ origen($p$) $\in$ computadoras(red($dcn$)) $\wedge_{\text{L}}$
destino($p$) $\in$ computadoras(red($dcn$)) $\wedge_{\text{L}}$ hayCamino?(red($dcn$), origen($p$), destino($p$))))
}
**Post** $\equiv$ {$dcn =_{\text{obs}}$ 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** $\equiv$ {$dcn =_{\text{obs}} dcn_0$}
**Post** $\equiv$ {$dcn =_{\text{obs}}$ 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($res =_{\text{obs}}$ 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($res =_{\text{obs}}$ 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** $\equiv$ {$res =_{\text{obs}}$ 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`) → *res* : `conj(paquete)`
**Pre** ≡ {c ∈ computadoras(red(*dcn*))}
**Post** ≡ {alias(*res* =$_{\text{obs}}$ 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*: `dcnet`, **in** *p*: `paquete`) → *res* : `bool`
**Pre** ≡ {true}
**Post** ≡ {*res* =$_{\text{obs}}$ 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*: `dcnet`) → *res* : `compu`
**Pre** ≡ {true}
**Post** ≡ {alias(*res* =$_{\text{obs}}$ laQueMasEnvio(*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*: $\texttt{dicc}_{trie}$`(puntero(compuDCNet))`,
*conjPaquetesDCNet*: `conj(paqueteDCNet)`,
*laQueMásEnvió*: `puntero(compuDCNet)`)

donde `compuDCNet` es `tupla`(*pc*: `puntero(compu)`,
*conjPaquetes*: `conj(paquete)`,
*diccPaquetesDCNet*: $\texttt{dicc}_{avl}$`(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) laQueMásEnvió 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_{\text{L}} \\
& (\forall c\text{: compu})(c \in \text{computadoras}(e.\text{topologia}) \Rightarrow \\
& ( \\
& (\exists cd\text{: compuDCNet})\ (\text{está?}(cd, e.\text{vectorCompusDCNet}) \wedge cd.\text{pc} = \text{puntero}(c)) \wedge \\
& (\exists s\text{: string})(\text{def?}(s, e.\text{diccCompusDCNet}) \wedge s = c.\text{ip}) \\
& ) \\
& ) \wedge_{\text{L}} \\
& (\forall cd\text{: compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \Rightarrow_{\text{L}} \\
& (\exists s\text{: string})\ (\text{def?}(s, e.\text{diccCompusDCNet}) \wedge \\
& s = cd.\text{pc} \rightarrow \text{ip} \wedge_{\text{L}} \text{obtener(s, } e.\text{diccCompusDCNet}) = \text{puntero}(cd)) \\
& ) \wedge_{\text{L}} \\
& (\exists cd\text{: compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \wedge_{\text{L}} \\
& *(cd.\text{pc}) = \text{compuQueMásEnvió}(e.\text{vectorCompusDCNet}) \wedge e.\text{laQueMásEnvió} = \text{puntero}(cd)) \wedge_{\text{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.\text{id} \neq p_2.\text{id}) \\
& ) \\
& ) \\
& ) \wedge_{\text{L}} \\
& (\forall cd\text{: compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \Rightarrow \\
& ( \\
& (\#(cd.\text{conjPaquetes}) = \#(\text{claves}(cd.\text{diccPaquetesDCNet}))) \wedge_{\text{L}} \\
& (\forall p\text{: paquete})(p \in cd.\text{conjPaquetes} \Rightarrow \\
& ( \\
& ((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.\text{pc})) \wedge_{\text{L}} \\
& (\exists sc\text{: secu(compu)})(sc \in \text{caminosMinimos}(e.\text{topologia}, p.\text{origen}, p.\text{destino}) \wedge \text{está}(*(cd.\text{pc}), sc))) \wedge \\
& (\exists n\text{: nat})\ ((\text{def?}(n, cd.\text{diccPaquetesDCNet}) \wedge p.\text{id} = n) \wedge_{\text{L}} \\
& (\text{Siguiente}(\text{obtener}(n, e.\text{diccPaquetesDCNet}).\text{it}) = p \wedge \\
& ((p.\text{origen} = *(cd.\text{pc}) \wedge \text{obtener}(n, e.\text{diccPaquetesDCNet}).\text{recorrido} = *(cd.\text{pc}) \bullet <>) \vee \\
& (p.\text{origen} \neq *(cd.\text{pc}) \wedge \\
& \text{obtener}(n, e.\text{diccPaquetesDCNet}).\text{recorrido} \in \text{caminosMinimos}(e.\text{topologia}, p.\text{origen}, *(cd.\text{pc})))) \\
& ) \\
& ) \wedge_{\text{L}} \\
& (\emptyset?(cd.\text{conjPaquetes}) \Leftrightarrow \text{vacía?}(cd.\text{colaPaquetesDCNet})) \wedge \\
& (\neg\text{vacía?}(cd.\text{colaPaquetesDCNet}) \Rightarrow_{\text{L}} \\
& (\exists n\text{: nat})(\text{def?}(n, cd.\text{diccPaquetesDCNet}) \wedge_{\text{L}} \\
& ( \\
& \text{Siguiente}(\text{obtener}(n, cd.\text{diccPaquetesDCNet}).\text{it}) = \text{paqueteMásPrioridad}(cd.\text{conjPaquetes}) \wedge \\
& \text{proximo}(cd.\text{colaPaquetesDCNet}) = \text{puntero}(\text{obtener}(n, cd.\text{diccPaquetesDCNet})) \\
& )) \\
& ) \wedge \\
& (cd.\text{enviados} \geq \text{enviadosCompu}(*(cd.\text{pc}), e.\text{vectorCompusDCNet})) \\
& ) \\
& )
\end{aligned}
$$

$\text{compuQueMásEnvió} : \text{secu(compuDCNet)}\ scd \longrightarrow \text{compu} \quad \{\neg\text{vacía?}(scd)\}$

$\text{maxEnviado} : \text{secu(compuDCNet)}\ scd \longrightarrow \text{nat} \quad \{\neg\text{vacía?}(scd)\}$

$\text{enviaronK} : \text{secu(compuDCNet)} \times \text{nat} \longrightarrow \text{conj(compu)}$

$\text{paqueteMásPrioridad} : \text{conj(paquete)}\ cp \longrightarrow \text{paquete} \quad \{\neg\emptyset?(cp)\}$

$\text{paquetesConPrioridadK} : \text{conj(cp)} \times \text{nat} \longrightarrow \text{conj(paquete)}$

$\text{altaPrioridad} : \text{conj(paquetes)}\ cp \longrightarrow \text{nat} \quad \{\neg\emptyset?(cp)\}$

$\text{enviadosCompu} : \text{compu} \times \text{secu(compuDCNet)} \longrightarrow \text{nat}$

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

compuQueMásEnvió($scd$) ≡ dameUno(enviaronK($scd$, maxEnviado($scd$)))
maxEnviado($scd$) ≡ **if** vacía?(fin($scd$)) **then** prim($scd$).enviados **else** max(prim($scd$), maxEnviado(fin($scd$))) **fi**
enviaronK($scd$, $k$) ≡ **if** vacía?($scd$) **then**
∅
**else**
**if** prim($scd$).enviados = $k$ **then**
Ag(*(prim($scd$).pc), enviaronK(fin($scd$), $k$))
**else**
enviaronK(fin($scd$), $k$)
**fi**
**fi**
paqueteMásPrioridad($dcn$, $cp$) ≡ 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 $e$ ⟶ dcnet {Rep($e$)}

Abs($e$) =$_{\text{obs}}$ dcn: dcnet | red($dcn$) = $e$.topología ∧
(∀$cdn$: compuDCNet)(está?($cdn$, $e$.vectorCompusDCNet) ⇒$_{\text{L}}$
enEspera($dcn$, *($cdn$.pc)) = $cdn$.conjPaquetes ∧
cantidadEnviados($dcn$, *($cdn$.pc)) = $cdn$.enviados ∧
(∀$p$: paquete)($p$ ∈ $cdn$.conjPaquetes ⇒$_{\text{L}}$
caminoRecorrido($dcn$, $p$) = obtener($p$.id, $cdn$.diccPaquetesDCNet).recorrido

)
)

# 2. Módulo Red

## 2.1. Interfaz

**se explica con**: RED.

**géneros**: `red`.

IniciarRed() → *res* : `red`
**Pre** ≡ {true}
**Post** ≡ {*res* =$_{\text{obs}}$ iniciarRed}
**Complejidad:** $O(1)$
**Descripción:** Crea una red nueva

AgregarComputadora(**in/out** $r$ : `red`, **in** $c$: `compu`)
**Pre** ≡ {$(r =_{\text{obs}} r_0) \wedge ((\forall\ c'$: compu) $(c' \in \text{computadoras}(r) \Rightarrow \text{ip}(c) \neq \text{ip}(c')))$ }
**Post** ≡ {$r =_{\text{obs}}$ 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** ≡ {$(r =_{\text{obs}} r_0) \wedge (c \in \text{computadoras}(r)) \wedge (c' \in \text{computadoras(r)}) \wedge (\text{ip}(c) \neq \text{ip}(c'))$
$\wedge$ (¬conectadas?($r$, $c$, $c'$)) $\wedge$ (¬usaInterfaz?($r$, $c$, $i$) $\wedge$ ¬usaInterfaz?($r$, $c'$, $i'$))}
**Post** ≡ {$r =_{\text{obs}}$ conectar($r_0$, $c$, $i$, $c'$, $i'$))}
**Complejidad:** $O(L)$?
**Descripción:** Conecta dos computadoras

Computadoras(**in** $r$ : `red`) → *res* : `conj(compu)`
**Pre** ≡ {true}
**Post** ≡ {alias($res =_{\text{obs}}$ computadoras($r$))}
**Complejidad:** $O(1)$
**Descripción:** Devuelve las computadoras de la red| [Devuelve una referancia no modificable]

Conectadas?(**in** $r$ : `red`, **in** $c$: `compu`, **in** $c'$ : `compu`) → *res* : `bool`
**Pre** ≡ {$(c \in \text{computadoras}(r)) \wedge (c' \in \text{computadoras}(r))$}
**Post** ≡ {$res =_{\text{obs}}$ 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`) → *res* : `interfaz`
**Pre** ≡ {conectadas?($r$, $c$, $c'$)}
**Post** ≡ {$res =_{\text{obs}}$ 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`) → *res* : `conj(compu)`
**Pre** ≡ {$c \in$ computadoras($r$)}
**Post** ≡ {$res =_{\text{obs}}$ 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

usaInterfaz?(**in** $r$ : `red`, **in** $c$ : `compu`, **in** $i$ : `interfaz`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {$c \in$ computadoras($r$)}
**Post** $\equiv$ {$res =_{\text{obs}}$ 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** $\equiv$ {alias($res =_{\text{obs}}$ caminosMinimos($r$, $c$, $i$))}
**Complejidad:** $O(L)$
**Descripción:** Devuelve el conjunto de caminos minimos de c a c'
**Aliasing:** Devuelve una refencia 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** $\equiv$ {$res =_{\text{obs}}$ 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** $\equiv$ {true}
**Post** $\equiv$ {$res =_{\text{obs}}$ $r$}
**Complejidad:** $O(?)$
**Descripción:** Devuelve una copia la red

$\bullet = \bullet$(**in** $r$ : `red`, **in** $r'$ : `red`) $\rightarrow$ $res$ : `bool`
**Pre** $\equiv$ {true}
**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 `tupla`(*compus*: `conj(compu)` ,
*dns*: $\text{dicc}_{Trie}$`(nodoRed)` )

donde `nodoRed` es `tupla`(*pc*: `puntero(compu)` ,
*caminos*: $\text{dicc}_{Trie}$`(conj(lista(compu)))` ,
*conexiones*: $\text{dicc}_{Lineal}$`(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 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`.*compus*), y los significados corresponden a todos los caminos mínimos desde la compu *pc* hacia la compu cuya IP es clave.

Rep : estr $\longrightarrow$ bool

Rep($e$) $\equiv$ true $\Longleftrightarrow$ (

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

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

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

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

((∀$i$: string, $n$: nodoRed) ((def?($i$, $e$.dns) $\wedge_{\text{L}}$ $n$ = obtener($i$, $e$.dns)) $\Rightarrow$
((∀$t$: nat) (def?($t$, $n$.conexiones) $\Rightarrow_{\text{L}}$ (obtener($t$, $n$.conexiones) $\neq$ puntero($n$))) )
)) $\wedge$

((∀$i$: string, $n$: nodoRed) ((def?($i$, $e$.dns) $\wedge_{\text{L}}$ $n$ = obtener($i$, $e$.dns)) $\Rightarrow$
((∀$t1, t2$: nat) (($t1 \neq t2 \wedge$ def?($t1$, $n$.conexiones) $\wedge$ def?($t2$, $n$.conexiones)) $\Rightarrow_{\text{L}}$
(obtener($t1$, $n$.conexiones) $\neq$ obtener($t2$, $n$.conexiones))
))
)) $\wedge$

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

)

| | | | | |
|---|---|---|---|---|
| vecinas | : nodoRed | $\longrightarrow$ | conj(nodoRed) | |
| auxVecinas | : nodoRed $\times$ dicc(nat $\times$ puntero(nodoRed)) | $\longrightarrow$ | conj(nodoRed) | |
| secusDeLongK | : conj(secu($\alpha$)) $\times$ nat | $\longrightarrow$ | conj(secu($\alpha$)) | |
| longMenorSec | : conj(secu($\alpha$)) *secus* | $\longrightarrow$ | nat | {¬∅?(*secus*)} |
| darRutas | : nodoRed $nA$ $\times$ nodoRed $nB$ $\times$ conj(pc) $\times$ secu(nodoRed) | $\longrightarrow$ | conj(secu(nodoRed)) | |
| darRutasVecinas | : conj(pc) *vec* $\times$ nodoRed $n$ $\times$ conj(pc) $\times$ secu(nodoRed) | $\longrightarrow$ | conj(secu(nodoRed)) | |
| darCaminosMinimos | : nodoRed $n1$ $\times$ nodoRed $n1$ | $\longrightarrow$ | conj(secu(compu)) | |

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

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

secusDeLongK($secus$, $k$) $\equiv$ **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($secus$) $\equiv$ **if** $\emptyset?$(sinUno($secus$)) **then**
long(dameUno($secus$))
**else**
min(long(dameUno($secus$)),
longMenorSec(sinUno($secus$)))
**fi**

darRutas($nA$, $nB$, $rec$, $ruta$) $\equiv$ **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($vecinas$, $n$, $rec$, $ruta$) $\equiv$ **if** $\emptyset?(vecinas)$ **then**
$\emptyset$
**else**
darRutas(dameUno($vecinas$), $n$, $rec$, $ruta$) $\cup$
darRutasVecinas(sinUno($vecinas$), $n$, $rec$, $ruta$)
**fi**

darCaminosMinimos($nA$, $nB$) $\equiv$ secusDeLongK(darRutas($nA$, $nB$, $\emptyset$, $<>$),
longMenorSec(darRutas($nA$, $nB$, $\emptyset$, $<>$))

#### 2.2.3. Función de Abstracción

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

Abs($e$) $=_{\text{obs}}$ r: red |

#### 2.2.4. Función de Abstracción

## 2.3. Algoritmos

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

```
iAgregarComputadora (in/out r : red, in c : compu)
  AgregoCompuNuevaAlResto( r.dns , c )                   O(n*L)
```

```
    AgregarRapido(r.compus, c)                                          O(1)
    Definir(r.dns, compu.ip, Tupla<&c, Vacio(), Vacio()>)               O(L)
    InicializarConjCaminos(r,c)                                         O(n)
```
**Complejidad** : $O(n * L)$

```
AgregoCompuNuevaAlResto (in/out r: red, in c: compu)
    itCompus:itConj(compu) ← CrearIt(r.compus)                          O(1)
    while HaySiguiente?(itCompus) do                                    O(1)
      nr:nodoRed ← Significado(r.dns,Siguiente(itCompus).ip)            O(L)
      Definir(nr.caminos, c.ip, Vacio())                                O(L)
      Avanzar(itCompus)                                                 O(1)
    end while                                                           O(n*L)
```
**Complejidad** : $O(n * L)$

```
InicializarConjCaminos (in/out r: red, in c: compu)
    itCompus:itConj(compu) ← CrearIt(r.compus)                          O(1)
    cams:diccTrie(ip,conj(lista(compu))) ←
     Significado(r.dns, c.ip).caminos                                   O(1)
    while HaySiguiente?(itCompus) do                                    O(1)
      Definir(cams, Siguiente(itCompus).ip, Vacio())                    O(L)
      Avanzar(itCompus)                                                 O(1)
    end while                                                           O(n)
```
**Complejidad** : $O(n)$

```
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(rn0.conexiones, i0, nr1)                              O(1)
    DefinirRapido(nr1.conexiones, i1, nr0)                              O(1)
    CrearCaminosDelDNS(r)                                               O(n!*(n^4))
```
**Complejidad** : $O(n! * (n^4))$

```
CrearCaminosDelDNS (in/out r: red)
    itCompus:itConj(compu) ← CrearIt(r.compus)                          O(1)
    while HaySiguiente?(itCompus) do                                    O(1)
      nr:nodoRed ← Significado(d,Siguiente(itCompus).ip)                O(L)
      AsignarCaminosMinimos(nr,r)                                       O(n!*(n^3))
      Avanzar(itCompus)                                                 O(1)
    end while                                                           O(n!*(n^4))
```
**Complejidad** : $O(n! * (n^4))$

```
AsignarCaminosMinimos (in/out nr: nodoRed)
    itCompus:itConj(compu) ← CrearIt(r.compus)                          O(1)
    while HaySiguiente?(itCompus) do                                    O(1)
      Definir(nr.caminos, Siguiente(itCompus).ip                        O(L)
        CrearCaminosMinimos(nr, Siguiente(itCompus).ip))                O(n!*(n^2))
      Avanzar(itCompus)                                                 O(1)
    end while                                                           O(n!*(n^3)
```
**Complejidad** : $O(n! * (n^3))$

```
CrearCaminosMinimos (in desde: nodoRed, in hasta: compu) → res: conj(lista(compu))
  camMins: conj(lista(compu)) ← Vacio()                                    O(1)
  itCamMins: itConj(conj(lista(compu))) ← CrearIt(camMins)                 O(1)
  pendientes: pila(nodoRed) ← Vacia()                                      O(1)
  ApilarConj(pendientes, Conexas(desde.conexiones))                        O(n)
  visitados: conj(nodoRed) ← Vacio()                                       O(1)
  cam: lista(compu) ← Vacia()                                              O(1)
  AgregarAdelante(cam, *(desde.pc))                                        O(1)
  while ¬EsVacia?(pendientes) do                                           O(1)
    AgregarRapido(visitados, Siguiente(itNrs))                             O(1)
    AgregarAdelante(cam, *(Siguiente(itNrs).pc))                           O(1)
    if (Tope(pendientes).pc→ip = hasta.ip) then                            O(L)
      if (Vacio?(camMins) ∨ (Longitud(cam) = Longitud(Siguiente
        (itCamMins)))) then                                                O(n)
        AgregarRapido(camMins,cam)                                         O(1)
      else
        if (Longitud(cam) < Longitud(Siguiente(itCamMins))) then           O(n)
          camMins ← Vacio()                                                O(1)
          AgregarRapido(camMins,cam)                                       O(1)
        endif                                                              O(n)
      endif                                                                O(n)
      Comienzo(cam)                                                        O(1)
    end if                                                                 O(n)
    Desapilar(pendientes)                                                  O(1)
    nv: conj(nodoRed) ← NoVisitadas(Tope(pendientes), visitados)           O(n^2)
    if(Vacia?(nv))                                                         O((1)
      Comienzo(cam)                                                        O(1)
    else
      ApilarConj(pendientes, nv)                                           O(n)
    end if                                                                 O(n)
  end while                                                                O(n! * n^2)
Complejidad : O(n! * (n^2))
```

```
NoVisitadas (in nr: nodoRed, in visitados: conj(nodoRed)) → res: conj(nodoRed)
  res ← Vacio()                                                            O(1)
  itVecinos: itConj(nodoRed) ← CrearIt(Conexas(nr.conexiones))             O(n)
  while HaySiguiente?(itVecinos) do                                        O(1)
    if(¬Pertenece?(visitados, Siguiente(itVecinos))) then                  O(n)
      AgregarRapido(res, Siguiente(itVecinos))                             O(1)
    end if                                                                 O(n)
    Avanzar(itVecinos)                                                     O(1)
  end while                                                                O(n^2)
Complejidad : O(n^2)
```

```
Conexas (in conex: dicc(interfaz,puntero(nodoRed))) → res: conj(nodoRed)
  res ← Vacio()                                                            O(1)
  itVecinos :itDicc(interfaz, puntero(nodoRed))) ← CrearIt(conex)          O(1)
  while HaySiguiente?(itVecinos) do                                        O(1)
    AgregarRapido(res, *SiguienteSignificado(itVecinos))                   O(1)
    Avanzar(itVecinos)                                                     O(1)
  end while                                                                O(n)
Complejidad : O(n)
```

```
ApilarConj (in/out pnr: pila(nodoRed), in news: conj(nodoRed))
  it: itConj(nodoRed) ← CrearIt(news)                                  O(1)
  while HaySiguiente?(it) do                                           O(1)
    Apilar(pnr, Siguiente(it))                                         O(1)
    Avanzar(it)                                                        O(1)
  end while                                                            O(n)
```
**Complejidad** : $O(n)$

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

```
iConectadas? (in r: red, in c0: compu, in c1: compu) → res: bool
  nr0: nodoRed ← Significado(r.dns, c0.ip)                             O(L)
  it : itDicc(interfaz, puntero(nodoRed))) ← CrearIt(nr0.conexiones)   O(1)
  res ← false                                                          O(1)
  while HaySiguiente?(it) do                                           O(1)
    if c1.ip = SiguienteSignificado(it)->pc->ip then                   O(1)
      res ← true                                                       O(1)
    end if                                                             O(1)
    Avanzar(it)                                                        O(1)
  end while                                                            O(n)
```
**Complejidad** : $O(L+n)$

```
iInterfazUsada (in r: red, in c0: compu, in c1: compu) → res: interfaz
  nr0: nodoRed ← Significado(r.dns, c0.ip)                             O(L)
  it : itDicc(itDicc(interfaz, puntero(nodoRed))
    ← CrearIt(nr0.conexiones)                                          O(1)
  while HaySiguiente?(it) do                                           O(1)
    if c1.ip = SiguienteSignificado(it)->pc->ip then                   O(1)
      res ← SiguienteClave(it)                                         O(1)
    end if                                                             O(1)
    Avanzar(it)                                                        O(1)
  end while                                                            O(n)
```
**Complejidad** : $O(L+n)$

```
iVecinos (in r: red, in c: compu) → res: conj(compu)
  nr: nodoRed ← Significado(r.dns, c.ip)                               O(L)
  res: conj(compu) ← Vacio()                                           O(1)
  it : itDicc(itDicc(interfaz, puntero(nodoRed))
    ← CrearIt(nr.conexiones)                                           O(1)
  while HaySiguiente?(it) do                                           O(1)
    AgregarRapido(res,*(SiguienteSignificado(it)->pc))                 O(1)
    Avanzar(it)                                                        O(1)
  end while                                                            O(n)
```
**Complejidad** : $O(L+n)$

```
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** $c_0$ : `compu`, **in** $c_1$ : `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** $c_0$ : `compu`, **in** $c_1$ : `compu`) → res: bool

```
nr:nodoRed ← Significado(r.dns, c0.ip)                    O(L)
res ← ¬EsVacio?(Significado(pnr.caminos, c1.ip))          O(L)
```

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

Copiar (**in** $r$ : `red`) → res: red

```
res:red ← iIniciarRed                                     O(1)
itCompus:itConj(compu) ← CrearIt(r.compus)                O(1)
 while HaySiguiente?(itCompus) do                         O(1)
  iAgregarComputadora(res, Siguiente(itCompus))           O(n*L)
  iConectar(res, Siguiente(itCompus))                     O(n!*(n^4))
end while                                                 O(n!*(n^5))
```

**Complejidad** : $O(n! * (n^5))$

$\bullet = \bullet$ (**in** $r_0$ : `red`, **in** $r_1$ : `red`) → res: bool

```
res ← (r0.compus = r1.compus) ∧ (r0.dns = r1.dns)         O(n^2)
```

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

# 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**

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

**parámetros formales**

**géneros** $\alpha$

**operaciones** $\bullet < \bullet$ : $\alpha \times \alpha \longrightarrow$ bool

Relación de orden total estricto[1]

**géneros** colaMinPrior($\alpha$)

**exporta** colaMinPrior($\alpha$), generadores, observadores

**usa** BOOL

**observadores básicos**

| | | | |
|---|---|---|---|
| vacía? | : colaMinPrior($\alpha$) | $\longrightarrow$ bool | |
| próximo | : colaMinPrior($\alpha$) $c$ | $\longrightarrow$ $\alpha$ | $\{\neg$ vacía?($c$)$\}$ |
| desencolar | : colaMinPrior($\alpha$) $c$ | $\longrightarrow$ colaMinPrior($\alpha$) | $\{\neg$ vacía?($c$)$\}$ |

**generadores**

| | | | |
|---|---|---|---|
| vacía | : | $\longrightarrow$ colaMinPrior($\alpha$) | |
| encolar | : $\alpha \times$ colaMinPrior($\alpha$) | $\longrightarrow$ colaMinPrior($\alpha$) | |

**otras operaciones**

| | | | |
|---|---|---|---|
| tamaño | : colaMinPrior($\alpha$) | $\longrightarrow$ nat | |

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

| | | |
|---|---|---|
| vacía?(vacía) | $\equiv$ | true |
| vacía?(encolar($e$, $c$)) | $\equiv$ | false |
| próximo(encolar($e$, $c$)) | $\equiv$ | **if** vacía?($c$) $\vee_{\text{L}}$ proximo($c$) $> e$ **then** $e$ **else** próximo($c$) **fi** |
| desencolar(encolar($e$, $c$)) | $\equiv$ | **if** vacía?($c$) $\vee_{\text{L}}$ proximo($c$) $> e$ **then** $c$ **else** encolar($e$, desencolar($c$)) **fi** |

**Fin TAD**

[1] 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* : `colaMinPrior(`$\alpha$`)`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {*res* $=_{\text{obs}}$ vacía}
**Complejidad:** O(1)
**Descripción:** Crea una cola de prioridad vacía

Vacía?(**in** *c*: `colaMinPrior(`$\alpha$`)`) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {*res* $=_{\text{obs}}$ vacía?(c)}
**Complejidad:** O(1)
**Descripción:** Devuelve `true` si y sólo si la cola está vacía

Desencolar(**in/out** *c*: `colaMinPrior(`$\alpha$`)`) $\rightarrow$ *res* : $\alpha$
**Pre** $\equiv$ {$\neg$vacía?($c$) $\wedge$ $c =_{\text{obs}} c_0$}
**Post** $\equiv$ {*res* $=_{\text{obs}}$ proximo($c_0$) $\wedge$ $c =_{\text{obs}}$ 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*: `colaMinPrior(`$\alpha$`)`, **in** $p$: `nat`, **in** $a$: $\alpha$)
**Pre** $\equiv$ {$c =_{\text{obs}} c_0$}
**Post** $\equiv$ {$c =_{\text{obs}}$ 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

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

donde `estr` es $\text{dicc}_{avl}$`(nat, 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 $\longrightarrow$ bool

Rep($e$) $\equiv$ true $\iff$
(∀$n$ : nat) def?($n$, $e$) $\Rightarrow_{\text{L}}$ ((obtener($n$, $e$).prioridad = $n$) ∧
¬vacía?(obtener($n$, $e$).encolados))

#### 3.3.3. Función de Abstracción

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

Abs($e$) $=_{\text{obs}}$ cmp: colaMinPrior | (vacía?($cmp$) $\Leftrightarrow$ (#claves($e$) = 0)) ∧
¬vacía?($cmp$) $\Rightarrow_{\text{L}}$
((próximo($cmp$) = próximo(mínimo($e$).encolados)) ∧
(desencolar($cmp$) = desencolar(mínimo($e$).encolados)))

### 3.4. Algoritmos

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

```
res ← Vacio()                                                    O(1)
```

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

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

```
res ← (#Claves(c) = 0)                                           O(1)
```

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

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

```
res ← Copiar(Proximo(Minimo(c).encolados))                       O(copy(α))
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** : $O(log(tamano(c)) + O(copy(\alpha))$

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

```
if Definido?(p) then                                             O(log(tamaño(c)))
    Encolar(Significado(c, p).encolados, a)                      O(log(tamaño(c)) + copy(α))
else
    nodoEncolados nuevoNodoEncolados                             O(1)
    nuevoNodoEncolados.encolados ← Vacia()                       O(1)
    nuevoNodoEncolados.prioridad ← p                             O(1)
    Encolar(nuevoNodoEncolados.encolados, a)                     O(copy(a))
    Definir(c, p, nuevoNodoEncolados)                            O(log(tamaño(c)) + copy(nodoEncolados))
end if
```

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

# 4. Módulo $\text{dicc}_{avl}(\alpha)$

## 4.1. Interfaz

**se explica con**: DICCIONARIO(NAT, $\alpha$).

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

### 4.1.1. Operaciones básicas de $\text{dicc}_{avl}(\alpha)$

CREARDICC() $\rightarrow$ *res* : $\texttt{dicc}_{avl}(\alpha)$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {*res* $=_{\text{obs}}$ vacío}
**Complejidad:** $O(1)$
**Descripción:** Crea un diccionario vacío

DEFINIDO?(**in** $c$: nat, **in** $d$: $\texttt{dicc}_{avl}(\alpha)$) $\rightarrow$ *res* : bool
**Pre** $\equiv$ {true}
**Post** $\equiv$ {*res* $=_{\text{obs}}$ def?($c$, $d$)}
**Complejidad:** $O(log(\#claves(d)))$
**Descripción:** Devuelve `true` si y sólo si la clave fue previamente definida en el diccionario

DEFINIR(**in** $c$: nat, **in** $s$: $\alpha$, **in/out** $d$: $\texttt{dicc}_{avl}(\alpha)$)
**Pre** $\equiv$ {$d =_{\text{obs}} d_0$}
**Post** $\equiv$ {$d =_{\text{obs}}$ definir($c$, $s$, $d_0$)}
**Complejidad:** $O(log(\#claves(d)) + copy(s))$
**Descripción:** Define la clave $c$ con el significado $s$ en $d$

OBTENER(**in** $c$: string, **in/out** $d$: $\texttt{dicc}_{avl}(\alpha)$) $\rightarrow$ *res* : $\alpha$
**Pre** $\equiv$ {def?($c$, $d$)}
**Post** $\equiv$ {alias(*res* $=_{\text{obs}}$ 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

MÍNIMO(**in/out** $d$: $\texttt{dicc}_{avl}(\alpha)$) $\rightarrow$ *res* : $\alpha$
**Pre** $\equiv$ {#claves($d$) >0}
**Post** $\equiv$ {alias(*res* $=_{\text{obs}}$ obtener(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

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

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

claveMínima($d$) $\equiv$ darClaveMínima($d$, claves($d$))

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

## 4.2. Representación

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

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

donde `estr` es `puntero(nodoAvl)`

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

# 5. Módulo Trie($\alpha$)

## 5.1. Interfaz

**se explica con**: DICCIONARIO(STRING, $\alpha$). **géneros**: $\texttt{dicc}_{Trie}(\alpha)$.

CREARDICC() $\rightarrow$ *res* : $\texttt{dicc}_{Trie}(\alpha)$
**Pre** $\equiv$ {true}
**Post** $\equiv$ {*res* $=_{\text{obs}}$ vacío}
**Complejidad:** $O(1)$
**Descripción:** Crea un diccionario vacío.

DEFINIDO?(**in** *c*: `string`, **in** *d*: $\texttt{dicc}_{Trie}(\alpha)$) $\rightarrow$ *res* : `bool`
**Pre** $\equiv$ {true}
**Post** $\equiv$ {*res* $=_{\text{obs}}$ 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*: $\texttt{dicc}_{Trie}(\alpha)$)
**Pre** $\equiv$ {$d =_{\text{obs}} d_0$}
**Post** $\equiv$ {$d =_{\text{obs}}$ 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*: $\texttt{dicc}_{Trie}(\alpha)$) $\rightarrow$ *res* : $\alpha$
**Pre** $\equiv$ {def?(*c*, *d*)}
**Post** $\equiv$ {alias(*res* $=_{\text{obs}}$ 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.