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`) $\rightarrow$ res : `conj(paquete)`
Pre $\equiv$ {c $\in$ computadoras(red(dcn))}
Post $\equiv$ {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`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {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`) $\rightarrow$ res : `compu`
Pre $\equiv$ {true}
Post $\equiv$ {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: `dicc`$_{trie}$`(puntero(compuDCNet))`,
conjPaquetesDCNet: `conj(paqueteDCNet)`,
laQueMásEnvió: `puntero(compuDCNet)`)

donde `compuDCNet` es `tupla`(*pc*: `puntero(compu)`,
conjPaquetes: `conj(paquete)`,
diccPaquetesDCNet: `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{array}{l}
\mathrm{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 \\
\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_{\text{L}} \\
(\forall cd\text{: compuDCNet})(\text{está?}(cd,\ e.\text{vectorCompusDCNet}) \Rightarrow_{\text{L}} \\
\quad (\exists s\text{: string})\ (\text{def?}(s,\ e.\text{diccCompusDCNet}) \wedge \\
\quad s = cd.\text{pc}{\to}\text{ip} \wedge_{\text{L}} \text{obtener}(\text{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 \\
\quad (\forall p_1\text{: paquete})(p_1 \in cd_1.\text{conjPaquetes} \Rightarrow \\
\quad\quad (\forall cd_2\text{: compuDCNet})((\text{está?}(cd_2,\ e.\text{vectorCompusDCNet}) \wedge cd_1 \neq cd_2) \Rightarrow \\
\quad\quad\quad (\forall p_2\text{: paquete})(p_2 \in cd_2.\text{conjPaquetes} \Rightarrow p_1.\text{id} \neq p_2.\text{id}) \\
\quad\quad) \\
\quad) \\
) \wedge_{\text{L}} \\
(\forall cd\text{: compuDCNet})(\text{está?}(cd,\ e.\text{vectorCompusDCNet}) \Rightarrow \\
\quad (\\
\quad (\#(cd.\text{conjPaquetes}) = \#(\text{claves}(cd.\text{diccPaquetesDCNet}))) \wedge_{\text{L}} \\
\quad (\forall p\text{: paquete})(p \in cd.\text{conjPaquetes} \Rightarrow \\
\quad\quad (\\
\quad\quad ((p.\text{origen} \in \text{computadoras}(e.\text{topologia}) \wedge p.\text{destino} \in \text{computadoras}(e.\text{topologia}) \wedge \\
\quad\quad p.\text{destino} \neq {*}(cd.\text{pc})) \wedge_{\text{L}} \\
\quad\quad (\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 \\
\quad\quad (\exists n\text{: nat})\ ((\text{def?}(n,\ cd.\text{diccPaquetesDCNet}) \wedge p.\text{id} = n) \wedge_{\text{L}} \\
\quad\quad (\text{Siguiente}(\text{obtener}(n,\ e.\text{diccPaquetesDCNet}).\text{it}) = p \wedge \\
\quad\quad ((p.\text{origen} = {*}(cd.\text{pc}) \wedge \text{obtener}(n,\ e.\text{diccPaquetesDCNet}).\text{recorrido} = {*}(cd.\text{pc}) \bullet <>) \vee \\
\quad\quad (p.\text{origen} \neq {*}(cd.\text{pc}) \wedge \\
\quad\quad \text{obtener}(n,\ e.\text{diccPaquetesDCNet}).\text{recorrido} \in \text{caminosMinimos}(e.\text{topologia},\ p.\text{origen},\ {*}(cd.\text{pc})))) \\
\quad\quad) \\
\quad) \wedge_{\text{L}} \\
\quad (\emptyset?(cd.\text{conjPaquetes}) \Leftrightarrow \text{vacía?}(cd.\text{colaPaquetesDCNet})) \wedge \\
\quad (\neg\text{vacía?}(cd.\text{colaPaquetesDCNet}) \Rightarrow_{\text{L}} \\
\quad\quad (\exists n\text{: nat})(\text{def?}(n,\ cd.\text{diccPaquetesDCNet}) \wedge_{\text{L}} \\
\quad\quad (\\
\quad\quad \text{Siguiente}(\text{obtener}(n,\ cd.\text{diccPaquetesDCNet}).\text{it}) = \text{paqueteMásPrioridad}(cd.\text{conjPaquetes}) \wedge \\
\quad\quad \text{proximo}(cd.\text{colaPaquetesDCNet}) = \text{puntero}(\text{obtener}(n,\ cd.\text{diccPaquetesDCNet})) \\
\quad\quad)) \\
\quad) \wedge \\
\quad (cd.\text{enviados} \geq \text{enviadosCompu}({*}(cd.\text{pc}),\ e.\text{vectorCompusDCNet})) \\
\quad) \\
)
\end{array}
$$

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

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

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

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

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

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

enviadosCompu : compu $\times$ secu(compuDCNet) $\longrightarrow$ nat aparicionesCompu : compu × conj(nat) cn × dicc(nat × paqueteDCNet) dp $\longrightarrow$ nat {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, dpd) $\equiv$ **if** $\emptyset$?(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 $\longrightarrow$ dcnet {Rep(e)}

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

)
)

2. Módulo Red

2.1. Interfaz

se explica con: RED.

géneros: `red`.

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

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

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

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

VECINOS(**in** r: `red`, **in** c: `compu`) $\to$ *res* : `conj(compu)`
Pre $\equiv$ $\{c \in \text{computadoras}(r)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{vecinos}(r, c)\}$
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 $\equiv$ $\{c \in \text{computadoras}(r)\}$
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 $\equiv$ $\{(c \in \text{computadoras}(r)) \wedge (c' \in \text{computadoras(r)})\}$
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 refencia no modificable

HAYCAMINO?(**in** r : `red`, **in** c : `compu`, **in** c' : `compu`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{(c \in \text{computadoras}(r)) \wedge (c' \in \text{computadoras(r)})\}$
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: $O(?)$
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: $O(?)$
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: `dicc`$_{Trie}$`(nodoRed)`)

donde `nodoRed` es `tupla`(*pc*: `puntero(compu)` ,
caminos: `dicc`$_{Trie}$`(conj(lista(compu)))` ,
conexiones: `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.

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

$\text{Rep}(e) \equiv \text{true} \iff ($

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

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

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

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

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

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

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

$)$

vecinas	: nodoRed	$\longrightarrow$ conj(nodoRed)	
auxVecinas	: nodoRed $\times$ dicc(nat $\times$ puntero(nodoRed))	$\longrightarrow$ conj(nodoRed)	
secusDeLongK	: conj(secu(α)) $\times$ nat	$\longrightarrow$ conj(secu(α))	
longMenorSec	: conj(secu(α)) *secus*	$\longrightarrow$ nat	$\{\neg\emptyset?(secus)\}$
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** $\emptyset?(cs)$ **then**
$\emptyset$
else
Ag(obtener(dameUno(claves(cs)), cs), auxVecinas(n, sinUno(cs)))
fi

secusDeLongK($secus$, k) ≡ **if** ∅?($secus$) **then**
  ∅
else
  if long(dameUno($secus$)) $= k$ **then**
    dameUno($secus$) ∪ secusDeLongK(sinUno($secus$), k)
  else
    secusDeLongK(sinUno($secus$), k)
  fi
fi

longMenorSec($secus$) ≡ **if** ∅?(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($ruta \circ nB$, ∅)
else
  if ∅?(vecinas(nA) - rec) **then**
    ∅
  else
    darRutas(dameUno(vecinas(nA) - rec),
    nB, Ag(nA, rec),
    $ruta \circ$ dameUno(vecinas(nA) - rec)) ∪
    darRutasVecinas(sinUno(vecinas(nA) - rec),
    nB, Ag(nA, rec),
    $ruta \circ$ dameUno(vecinas(nA) - rec))
  fi
fi

darRutasVecinas($vecinas$, n, rec, $ruta$) ≡ **if** ∅?($vecinas$) **then**
  ∅
else
  darRutas(dameUno($vecinas$), n, rec, $ruta$) ∪
  darRutasVecinas(sinUno($vecinas$), n, rec, $ruta$)
fi

darCaminosMinimos(nA, nB) ≡ secusDeLongK(darRutas(nA, nB, ∅, $<>$),
longMenorSec(darRutas(nA, nB, ∅, $<>$)))

2.2.3. Función de Abstracción

Abs : estr e ⟶ red {Rep(e)}

Abs(e) $=_{\text{obs}}$ r: red | e.compus $=_{\text{obs}}$ computadoras(r) ∧
((∀ $c1$, $c2$: compu, $i1$, $i2$: string, $n1$, $n2$: nodoRed) (
($c1 \in e$.compus ∧ $i1 = c1$.ip ∧ def?($i1$, e.dns) $\wedge_L$ $n1 =$ obtener($i1$, e.dns) ∧ $c1 = {*}n1$.pc) ∧
($c2 \in e$.compus ∧ $i2 = c2$.ip ∧ def?($i2$, e.dns) $\wedge_L$ $n2 =$ obtener($i2$, e.dns) ∧ $c2 = {*}n2$.pc) ∧
($c1 \neq c2$))) $\Rightarrow_L$
(conectadas?(r, $c1$, $c2$) ⇔ ((∃ $t1$, $t2$: nat) (
$t1 =$ interfazUsada(r, $c1$, $c2$) ∧ $t2 =$ interfazUsada(r, $c2$, $c1$) ∧
def?($t1$, $n1$.conexiones) ∧ def?($t2$, $n2$.conexiones) $\wedge_L$ (
&$n2 =$ obtener($t1$, $n1$.conexiones) ∧ &$n1 =$ obtener($t2$, $n2$.conexiones)
))))

3. Módulo Cola de mínima prioridad(α)

El módulo cola de mínima prioridad consiste en una cola de prioridad de elementos del tipo α 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(α)

igualdad observacional

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

parámetros formales

géneros α

operaciones • $<$: $\alpha \times \alpha \longrightarrow$ bool   Relación de orden total estricto[1]

•

géneros colaMinPrior(α)

exporta colaMinPrior(α), generadores, observadores

usa BOOL

observadores básicos

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

generadores

vacía	:	$\longrightarrow$	colaMinPrior(α)
encolar	: $\alpha \times$ colaMinPrior(α)	$\longrightarrow$	colaMinPrior(α)

otras operaciones

tamaño	: colaMinPrior(α)	$\longrightarrow$	nat

axiomas $\forall$ c: colaMinPrior(α), $\forall$ e: α

vacía?(vacía)	$\equiv$	true
vacía?(encolar(e, c))	$\equiv$	false
próximo(encolar(e, c))	$\equiv$	**if** vacía?(c) $\lor_{\text{L}}$ proximo(c) $> e$ **then** e **else** próximo(c) **fi**
desencolar(encolar(e, c))	$\equiv$	**if** vacía?(c) $\lor_{\text{L}}$ 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: $((a < b \land b < c) \;\Rightarrow\; a < c)$ para todo $a, b, c : \alpha$

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

3.2. Interfaz

parámetros formales
géneros α

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

géneros: `colaMinPrior`(α).

3.2.1. Operaciones básicas de Cola de mínima prioridad

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

VACÍA?(**in** c: `colaMinPrior`(α)) $\rightarrow$ *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`(α)) $\rightarrow$ *res* : α
Pre $\equiv$ $\{\neg$vacía?$(c) \land c =_{\text{obs}} c_0\}$
Post $\equiv$ $\{res =_{\text{obs}}$ proximo$(c_0) \land c =_{\text{obs}}$ 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`(α), **in** p: `nat`, **in** a: α)
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 α con prioridad p a la cola
Aliasing: Se agrega el elemento por copia

3.3. Representación

3.3.1. Representación de colaMinPrior

`colaMinPrior`(α) **se representa con** `estr`

donde `estr` es `dicc`$_{avl}$(`nat, nodoEncolados`)

donde `nodoEncolados` es `tupla`(*encolados*: `cola`(α), *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 $\Longleftrightarrow$
($\forall n$: nat) def?(n, e) $\Rightarrow_L$ ((obtener(n, e).prioridad $=$ n) $\wedge$ $\neg$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)) $\wedge$
$\neg$vacía?(cmp) $\Rightarrow_L$
((próximo(cmp) $=$ próximo(mínimo(e).encolados)) $\wedge$
(desencolar(cmp) $=$ desencolar(mínimo(e).encolados)))

3.4. Algoritmos

iVacía () $\to$ res: colaMinPrior(α)

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

Complejidad : $O(1)$

iVacía? (**in** c: `colaMinPrior`(α)) $\to$ res: bool

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

Complejidad : $O(1)$

iDesencolar (**in/out** c: `colaMinPrior`(α)) $\to$ res: α

```
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`(α), **in** p: `nat`, **in** a: α)

```
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 Diccionario AVL(α)

4.1. Interfaz

se explica con: Diccionario(nat, α).

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

4.1.1. Operaciones básicas de Diccionario AVL(α)

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: α, **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* : α
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* : α
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 $\times$ α) d	$\longrightarrow$ nat	{#claves(d) >0}
darClaveMínima	:	dicc(nat $\times$ α) 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 dicc$_{avl}(\alpha)$

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

donde `estr` es `puntero(nodoAvl)`

donde `nodoAvl` es `tupla(` *clave*: `nat`, *data*: α, *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 *balance* $\in$ {-1, 0, 1 } donde *balance* es:
* 0 si el arbol esta balanciado
* 1 si existe un nodo en el ultimo nivel de balance tal que tenga un hijo a la izq
* -1 si existe un nodo en el ultimo nivel de balance tal que tenga un hijo a la der

(III) estr esta balaciado entero o podado (quitar le las hojas del ultimo nivel) esta balaciado

(IV) Todas las claves son distintas.

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

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

esABB	: puntero(nodoAvl)	$\longrightarrow$	bool
balanceadoBien	: puntero(nodoAvl)	$\longrightarrow$	bool
clavesDistinta	: puntero(nodoAvl) $\times$ conj(nat)	$\longrightarrow$	bool
balanceado	: puntero(nodoAvl)	$\longrightarrow$	bool
podar	: puntero(nodoAvl)	$\longrightarrow$	bool

esABB(n) $\equiv$ $(n \neq \text{NULL}) \Rightarrow_{\text{L}}$ (
((prim($n\to$hijos) $\neq$ NULL) $\Rightarrow_{\text{L}}$ ($n\to$clave $>$ prim(n.hijos) $\land$ esABB(prim($n\to$hijos)))) $\land$
((ult($n\to$hijos) $\neq$ NULL) $\Rightarrow_{\text{L}}$ ($n\to$clave $<$ ult($n\to$hijos) $\land$ esABB(ult($n\to$hijos)))))

balanciadoBien(n) $\equiv$ (balanceado(n) $\lor$ balanceado(podar(n)) $\land_{\text{L}}$ $(n \neq \text{NULL}) \Rightarrow_{\text{L}}$ (
if ((prim($n\to$hijos) $\neq$ NULL) $\land$ (ult($n\to$hijos) $\neq$ NULL)) **then**
balanceadoBien(prim($n\to$hijos)) $\land$ balanceadoBien(ult($n\to$hijos)))
else
if (prim(n.hijos) $\neq$ NULL) **then**
$n\to$balance = 1
else
if (prim(n.hijos) $\neq$ NULL) **then** $n\to$balance = -1 **else** $n\to$balance = 0 **fi**
fi
fi

clavesDistintas(n, cs) $\equiv$ $(n \neq \text{NULL}) \Rightarrow_{\text{L}}$
$n\to$clave $\notin cs$ $\land$
clavesDistintas(prim($n\to$hijos), Ag($n\to$clave, cs)) $\land$
clavesDistintas(ult($n\to$hijos), Ag($n\to$clave, cs))

balanceado(n) $\equiv$ ($n \neq$ NULL) $\Rightarrow_L$
(**if** ((prim($n \rightarrow$hijos) $\neq$ NULL) $\wedge$ (ult($n \rightarrow$hijos) $\neq$ NULL)) **then**
  balanceado(prim($n \rightarrow$hijos)) $\wedge$ balanceado(ult($n \rightarrow$hijos)))
else
  if (prim(n.hijos) $\neq$ NULL) **then**
    false
  else
    if (prim(n.hijos) $\neq$ NULL) **then** false **else** true **fi**
  fi
fi)

4.2.3. Función de Abstracción

Abs : estr e $\longrightarrow$ dicc(nat, α) {Rep(e)}

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

auxAbs : puntero(nodoAvl) $\times$ dicc(nat; α) $\longrightarrow$ dicc(nat, α)

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

4.3. Algoritmos

iInsertar (**in/out** *puntero(nodoAvl)*: `tree`, **in** *nat*: `c`, **in** α: `s`)

```
if (tree->root == NULL) then
{
    tree->root = make_node(data);

    if (tree->root == NULL)
    {
        return 0;
    }
}
```

Complejidad : $O(1)$

iVacío () → res: puntero(nodoAvl)

```
res ← NULL                                                        O(1)
```

Complejidad : $O(1)$

iDefinir (**in/out** p: `puntero(nodoAvl)`, **in/out** *clave*: `nat`, **in/out** *significado*: `alpha`)

```
padre: puntero(nodoAvl) ← p                                       O(1)
nuevo: nodoAvl ← {clave: clave, data: significado, altura: 0, hijos: [NULL, NULL]}  O(copy(significado))
bool: buscarMas ← true                                            O(1)
dir: int

while (buscarMas) do                                              O(1)
    if (padre.clave = clave) then                                 O(1)
        padre.data ← significado                     O(copy(significado))
        buscarMas ← false                                         O(1)
```

```
        else
            if (padre.clave > clave) then                                  O(1)
                dir ← 0                                                    O(1)
            else
                dir ← 1                                                    O(1)
            end if

            if (padre.hijos[dir] = NULL) then                              O(1)
                padre.hijos[dir] ← &nuevo                                  O(1)
                balancear(p)
                buscarMas ← false                                          O(1)
            else
                padre ← padre.hijos[dir]                                   O(1)
            end if

        end if
    end while                                                              O(p.altura)
```
Complejidad : $O(p.altura + copy(significado))$

iBorrar () → res: puntero(nodoAvl)

```
    res ← NULL                                                             O(1)
```
Complejidad : $O(1)$

iDefinido? () → res: puntero(nodoAvl)

```
    res ← NULL                                                             O(1)
```
Complejidad : $O(1)$

iObtener () → res: puntero(nodoAvl)

```
    res ← NULL                                                             O(1)
```
Complejidad : $O(1)$

iClaves () → res: puntero(nodoAvl)

```
    res ← NULL                                                             O(1)
```
Complejidad : $O(1)$

Mínimo (**in** $dicc_{avl}(\alpha)$: d) → res: α

```
    actual:puntero(nodoAvl) ← d
    hijoMenor:puntero(nodoAvl)
    done:bool ← false

    while (!done) do
        hijoMenor ← (actual→hijos[0])

        if (hijoMenor != NULL) then
            actual ← hijoMenor
        else
            res ← (actual→data)
            done ← true
        end if
```

```
end while
```

Inorder (**in** $dicc_{avl}(\alpha)$: `n`) $\rightarrow$ res: lista(tupla(clave, significado))

```
c:puntero(nodoAvl) ← n
p:pila(puntero(nodoAvl)) ← Vacia()
done:bool ← false
res ← Vacia()

while (!done) do
    if (c != NULL) then
        Apilar(p, c)
        c ← (c→hijos[0])
    else
        if !EsVacia?(p) then
            AgregarAtras(res, << Tope(p)->clave, Tope(p)->data >>)
            c ← Tope(p)→hijos[1]
        else
            done ← true
        end if
    end if
end while
```

$\bullet = \bullet$ (**in** $dicc_{avl}(\alpha)$: `d1`, **in** $dicc_{avl}(\alpha)$: `d2`) $\rightarrow$ res: bool

```
res ← Inorder(d1) = Inorder(d2)
```

5. Módulo Trie(α)

5.1. Interfaz

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

CREARDICC() $\rightarrow res$: $\texttt{dicc}_{Trie}(\alpha)$
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{vacío}\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{def?}(c, d)\}$
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: α, **in/out** d: $\texttt{dicc}_{Trie}(\alpha)$)
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: $\texttt{dicc}_{Trie}(\alpha)$) $\rightarrow res$: α
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.