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`) → res : `dcnet`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ 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 ≡ {
$dcn =_{\text{obs}} dcn_0$ ∧
¬((∃ p':paquete)(paqueteEnTransito(dcn, p') ∧ id(p) = id(p') ∧ origen(p) ∈ computadoras(red(dcn)) $\land_{\text{L}}$ destino(p) ∈ computadoras(red(dcn)) $\land_{\text{L}}$ hayCamino?(red(dcn), origen(p), destino(p))))
}
Post ≡ {$dcn =_{\text{obs}}$ 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 ≡ {$dcn =_{\text{obs}} dcn_0$}
Post ≡ {$dcn =_{\text{obs}}$ 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`) → res : `red`
Pre ≡ {true}
Post ≡ {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`) → res : `secu(compu)`
Pre ≡ {paqueteEnTransito?(dcn, p)}
Post ≡ {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`) → res : `nat`
Pre ≡ {c ∈ computadoras(red(dcn))}
Post ≡ {$res =_{\text{obs}}$ 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

$\bullet = \bullet$(**in** dcn_1: `dcnet`, **in** dcn_2: `dcnet`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} dcn_1 = dcn_2$}
Complejidad: $O(n * k^3 * (k + n))$
Descripción: compara dcn_1 y dcn_2 por igualdad

1.2. Representación

1.2.1. Representación de dcnet

`dcnet` **se representa con** `estr`

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

donde `compuDCNet` es `tupla`(*pc*: `puntero(compu)`,
conjPaquetes: `conj(paquete)`,
diccPaquetesDCNet: $\texttt{dicc}_{avl}$`(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) El conjPaquetesDCNet contiene tuplas con iteradores a todos los paquetes en tránsito en la red y sus recorridos

(VI) Todos los paquetes en conjPaquetes de cada compuDCNet tienen id único y tanto su origen como destino existen en la topología

(VII) El paquete en conjPaquetes tiene que tener en su recorrido a la compuDCNet en la que se encuentra y esta no puede ser igual al destino del recorrido

(VIII) Las claves de diccPaquetesDCNet son los id de los paquetes en conjPaquetes

(IX) Los significados de diccPaquetesDCNet son un iterador al paqueteDCNet de conjPaquetesDCNet que contiene un iterador al paquete con el id equivalente a su clave y un recorrido que es uno de los caminos mínimos del origen del paquete a la compu en la que se encuentra

(X) La cantidad de enviados de una compuDCNet es igual o mayor a la cantidad de apariciones de esa compu en los caminos recorridos de paquetes en la red

(XI) El paquete a enviar de cada compuDCNet es un iterador que no tiene siguiente

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
(∀c: compu)($c \in$ computadoras(e.topologia) ⇔
(
(∃cd: compuDCNet) (está?(cd, e.vectorCompusDCNet) ∧ (cd.pc = puntero(c)) ∧
(∃s: string)(def?(s, e.diccCompusDCNet) ∧ ($s = c$.ip)))
)
) $\wedge_L$
(∀cd: compuDCNet)(está?(cd, e.vectorCompusDCNet) ⇔
(∃s: string)(($s = cd$.pc→ip) ∧ def?(s, e.diccCompusDCNet) $\wedge_L$
obtener(s, e.diccCompusDCNet) = puntero(cd))
) $\wedge_L$
(∃cd: compuDCNet)(está?(cd, e.vectorCompusDCNet) $\wedge_L$
*(cd.pc) = compuQueMásEnvió(e.vectorCompusDCNet) ∧ e.laQueMásEnvió = puntero(cd)) $\wedge_L$
(∀cd_1: compuDCNet)(está?(cd_1, e.vectorCompusDCNet) ⇒
(∀p_1: paquete)($p_1 \in cd_1$.conjPaquetes ⇒
(∀cd_2: compuDCNet)((está?(cd_2, e.vectorCompusDCNet) ∧ $cd_1 \neq cd_2$) ⇒
(∀p_2: paquete)($p_2 \in cd_2$.conjPaquetes ⇒ p_1.id $\neq$ p_2.id)
)
)
) $\wedge_L$
(∀cd: compuDCNet)(está?(cd, e.vectorCompusDCNet) ⇒
(
(∀p: paquete)($p \in cd$.conjPaquetes ⇔
(
((p.origen ∈ computadoras(e.topologia) ∧ p.destino ∈ computadoras(e.topologia) ∧
p.destino $\neq$ *(cd.pc)) $\wedge_L$
(∃sc: secu(compu))(sc ∈ caminosMinimos(e.topologia, p.origen, p.destino) ∧ está(*(cd.pc), sc))) ∧
(∃n: nat) ((def?(n, cd.diccPaquetesDCNet) ∧ p.id = n) $\wedge_L$
(∃pdn: paqueteDCNet)(pdn ∈ e.conjPaquetesDCNet ∧ Siguiente(pdn.it) = p ∧
((p.origen = *(cd.pc) ∧ pdn.recorrido = *(cd.pc) • <>) ∨
(p.origen $\neq$ *(cd.pc) ∧ pdn.recorrido ∈ caminosMinimos(e.topologia, p.origen, *(cd.pc)))) ∧
Siguiente(obtener(n, cd.diccPaquetesDCNet)) = pdn
)
)
)
) $\wedge_L$
(¬vacía?(cd.colaPaquetesDCNet) ⇔
(∃p: paquete)(($p \in cd$.conjPaquetes) ∧ (p = paqueteMásPrioridad(cd.conjPaquetes)) ∧
(∃pdn: paqueteDCNet)((pdn ∈ e.conjPaquetesDCNet) ∧ (Siguiente(pdn.it) = p) ∧
(Siguiente(proximo(cd.colaPaquetesDCNet)) = pdn))
)
) $\wedge_L$
(cd.enviados ≥ enviadosCompu(*(cd.pc), e.vectorCompusDCNet)) ∧
(¬HaySiguiente?(cd.paqueteAEnviar)))
)

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

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

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

paqueteMásPrioridad : conj(paquete) cp $\longrightarrow$ paquete {¬∅?(cp)}

paquetesConPrioridadK : conj(paquete) × nat $\longrightarrow$ conj(paquete)

altaPrioridad : conj(paquetes) cp $\longrightarrow$ nat {¬∅?(cp)}

enviadosCompu : compu × secu(compuDCNet) $\longrightarrow$ nat

aparicionesCompu : compu × conj(nat) cn × dicc(nat × itConj(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, Siguiente(obtener(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 ∧
(∀cdn: compuDCNet)(está?(cdn, e.vectorCompusDCNet) $\Rightarrow_{\text{L}}$
enEspera(dcn, *(cdn.pc)) = cdn.conjPaquetes ∧
cantidadEnviados(dcn, *(cdn.pc)) = cdn.enviados ∧
(∀p: paquete)(p ∈ cdn.conjPaquetes $\Rightarrow_{\text{L}}$
caminoRecorrido(dcn, p) = Siguiente(obtener(p.id, cdn.diccPaquetesDCNet)).recorrido
)
)

1.3. Algoritmos

iIniciarDCNet (**in** *topo* : `red`) → res: estr

```
res.topologia ← Copiar(topo)                                          O(n! * n^6)
res.vectorCompusDCNet ← Vacia()                                       O(1)
res.diccCompusDCNet ← CrearDicc()                                     O(1)
res.laQueMasEnvio ← NULL                                              O(1)
res.conjPaquetesDCNet ← Vacio()                                       O(1)

itConj(compu): it ← CrearIt(Computadoras(topo))                       O(1)

if(HaySiguiente?(it)) then                                            O(1)
    res.laQueMasEnvio ← puntero(Siguiente(it))                        O(1)
end if

while HaySiguiente?(it) do                                            O(1)
    compuDCNet: compudcnet ← <puntero(Siguiente(it)), Vacio(), CrearDicc(),
        Vacia(), CrearIt(Vacio()), 0>                                 O(1)
    AgregarAtras(res.vectorCompusDCNet, compudcnet)                   O(n)
    Definir(res.diccCompusDCNet, Siguiente(it).ip, puntero(compudcnet))  O(L)
    Avanzar(it)                                                       O(1)
end while                                                             O(n * (n + L))
```

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

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

```
puntero(compuDCNet): compudcnet ←
    Significado(dcn.diccCompusDCNet, p.origen.ip)                     O(L)
itConj(paquete): itPaq ← AgregarRapido(compudcnet→conjPaquetes, p)    O(1)
lista(compu): recorr ← AgregarAtras(Vacia(), p.origen)                O(1)
paqueteDCNet: paqDCNet ← <itPaq, recorr>                              O(1)

itConj(paqueteDCNet): itPaqDCNet ←
    AgregarRapido(dcn.conjPaquetesDCNet, paqDCNet)                    O(1)
Definir(compudcnet→diccPaquetesDCNet, itPaqDCNet)                     O(log(k))
Encolar(compudcnet→colaPaquetesDCNet, itPaqDCNet)                     O(log(k))
```

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

iAvanzarSegundo (**in/out** *dcn*: `dcnet`)

```
nat: maxEnviados ← 0
nat: i ← 0                                                               O(1)
while i < Longitud(dcn.vectorCompusDCNet) do                             O(1)
    if(¬EsVacia?(dcn.vectorCompusDCNet[i].colaPaquetesDCNet)) then
        dcn.vectorCompusDCNet[i].paqueteAEnviar ←
            Desencolar(dcn.vectorCompusDCNet[i].colaPaquetesDCNet)       O(log(k))
    end if
    i++                                                                  O(1)
end while                                                                O(n * log(k))

i ← 0                                                                    O(1)
while i < Longitud(dcn.vectorCompusDCNet) do                             O(1)
    if(HaySiguiente?(dcn.vectorCompusDCNet[i].paqueteAEnviar)) then      O(1)

        dcn.vectorCompusDCNet[i].enviados++                              O(1)
        if(dcn.vectorCompusDCNet[i].enviados > maxEnviados) then         O(1)
            dcn.laQueMasEnvio ← puntero(dcn.vectorCompusDCNet[i])        O(1)
        end if

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

        if(pAEnviar.destino ≠ siguientecompu) then                       O(1)

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

            itConj(paquete): itpaquete ←
                AgregarRapido(siguientecompudcnet.conjPaquetes, pAEnviar)    O(1)

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

            AgregarAtras(Siguiente(paqAEnviar).recorrido, siguientecompu)    O(1)

            Encolar(siguientecompudcnet.colaPaquetesDCNet, paqAEnviar)   O(log(k))
            Definir(siguientecompudcnet.diccPaquetesDCNet, pAEnviar.id,
                paqAEnviar)                                              O(log(k))
        end if

        Borrar(dcn.vectorCompusDCNet[i].diccPaquetesDCNet,
            Siguiente(dcn.vectorCompusDCNet[i].paqueteAEnviar→it).id)    O(log(k))
        EliminarSiguiente(Siguiente(dcn.vectorCompusDCNet[i].paqueteAEnviar).it)
                                                                         O(1)
        EliminarSiguiente(dcn.vectorCompusDCNet[i].paqueteAEnviar)       O(1)

        dcn.vectorCompusDCNet[i].paqueteAEnviar ← CrearIt(Vacio())       O(1)

    end if
    i++                                                                  O(1)
end while                                                                O(n * (L + log(k)))
```

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

Red (**in** dcn : `dcnet`) → res: red

```
res ← dcn.topologia                                                    O(1)
```

Complejidad : $O(1)$

CaminoRecorrido (**in** dcn : `dcnet`, **in** p : `paquete`) → res: lista(compu)

```
nat: i ← 0                                                             O(1)
while i < Longitud(dcn.vectorCompusDCNet) do                           O(1)
    if Definido?(dcn.vectorCompusDCNet[i].diccPaquetesDCNet, p.id) then O(log(k))
        res ← Siguiente(Obtener(dcn.vectorCompusDCNet[i].diccPaquetesDCNet,
            p.id)).recorrido                                           O(log(k))
    end if
    i++                                                                O(1)
end while                                                              O(n * log(k))
```

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

CantidadEnviados (**in** dcn : `dcnet`, **in** c : `compu`) → res: nat

```
res ← Obtener(dcn.diccCompusDCNet, c.ip)→enviados                      O(L)
```

Complejidad : $O(L)$

EnEspera (**in** dcn : `dcnet`, **in** c : `compu`) → res: nat

```
res ← Obtener(dcn.diccCompusDCNet, c.ip)→conjPaquetes                  O(L)
```

Complejidad : $O(L)$

PaqueteEnTransito (**in** dcn : `dcnet`, **in** p : `paquete`) → res: bool

```
res ← false
nat: i ← 0                                                             O(1)
while i < Longitud(dcn.vectorCompusDCNet) do                           O(1)
    if Definido?(dcn.vectorCompusDCNet[i].diccPaquetesDCNet, p.id) then O(log(k))
        res ← true                                                     O(1)
    end if
    i++                                                                O(1)
end while                                                              O(n * log(k))
```

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

LaQueMasEnvio (**in** dcn : `dcnet`) → res: compu

```
res ← *(dcn.laQueMasEnvio→pc)                                                     O(1)
```

Complejidad : $O(1)$

• $=_i$ • (**in** dcn_1 : `dcnet`, **in** dcn_2 : `dcnet`) → res: bool

```
bool: boolTopo ← dcn1.topologia = dcn2.topologia                                  O(n + L^2)
bool: boolVec ← dcn1.vectorCompusDCNet = dcn2.vectorCompusDCNet           O(n * k * (k + n))
bool: boolConj ← dcn1.conjPaquetesDCNet = dcn2.conjPaquetesDCNet            O(k^3 * (k + n))
bool: boolMasEnvio ← dcn1→ =paqdcn dcn2→                                          O(1)

res ← boolTopo ∧ boolVec ∧ boolTrie ∧ boolConj ∧ boolMasEnvio                     O(1)
```

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

• $=_{compudcn}$ • (**in** c_1 : `compuDCNet`, **in** c_2 : `compuDCNet`) → res: bool

```
bool: boolPC ← *(c1.pc) = *(c2.pc)                                                O(1)
bool: boolConj ← c1.conjPaquetes = c1.conjPaquetes                                O(k^2)
bool: boolAVL ← true                                                              O(1)
bool: boolCola ← true                                                             O(1)
bool: boolPaq ← Siguiente(c1.paqueteAEnviar) =paqdcn Siguiente(c2.paqueteAEnviar)
                                                                                  O(n)
bool: boolEnviados ← c1.enviados = c2.enviados                                    O(1)

if boolConj then                                                                  O(1)
    itConj: itconj1 ← CrearIt(c1.conjPaquetes)                                    O(1)
    while HaySiguiente?(itconj1) do                                               O(1)
        if Definido?(c2.diccPaquetesDCNet, Siguiente(itconj1)).id then       O(log(n))
            if ¬(Siguiente(Obtener(c1.diccPaquetesDCNet, Siguiente(itconj1).id))
               =paqdcn
               Siguiente(Obtener(c1.diccPaquetesDCNet, Siguiente(itconj1).id)))
               then                                                               O(n)
               boolAVL ← false                                                    O(1)
            end if
        else
            boolAVL ← false                                                       O(1)
        end if
        Avanzar(itconj1)                                                          O(1)
    end while                                                                 O(n * k)
end if

if EsVacia(c1.colaPrioridad) then                                                 O(1)
    if ¬EsVacia(c2.colaPrioridad) then                                            O(1)
        boolCola ← false                                                          O(1)
    end if
else
    if EsVacia(c1.colaPrioridad) then                                             O(1)
        boolCola ← false                                                          O(1)
    else
        if ¬(Siguiente(Proximo(c1.colaPrioridad)) =paqdcn
            Siguiente(Proximo(c2.colaPrioridad))) then                            O(n)
            boolCola ← false                                                      O(1)
        end if
```

```
        end if
    end if

    res ← boolPC ∧ boolConj ∧ boolAVL ∧ boolCola ∧ boolPaq ∧ boolEnviados       O(1)
```

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

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

```
    bool: boolPaq ← Siguiente(p1.it) = Siguiente(p2.it)       O(1)
    bool: boolRecorrido ← p1.recorrido = p2.recorrido       O(n)

    res ← boolPaq ∧ boolRecorrido       O(1)
```

Complejidad : $O(n)$

2. Módulo Red

2.1. Interfaz

se explica con: RED.

géneros: `red`.

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

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

CONECTAR(**in/out** r : `red`, **in** c : `compu`, **in** c' : `compu`, **in** i : `compu`, **in** i' : `compu`)
Pre $\equiv$ $\{(r =_{\text{obs}} r_0) \wedge (c \in$ 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 $\equiv$ $\{r =_{\text{obs}}$ conectar$(r_0, c, i, c', i'))\}$
Complejidad: $O(n! * (n^4))$
Descripción: Conecta dos computadoras

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

$\text{USAINTERFAZ?}(\textbf{in } r\colon \texttt{red}, \textbf{in } c\colon \texttt{compu}, \textbf{in } i\colon \texttt{interfaz}) \to res : \texttt{bool}$
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

$\text{CAMINOSMINIMOS}(\textbf{in } r\colon \texttt{red}, \textbf{in } c\colon \texttt{compu}, \textbf{in } c'\colon \texttt{compu}) \to res : \texttt{conj(lista(compu))}$
Pre $\equiv \{(c \in \text{computadoras}(r)) \land (c' \in \text{computadoras(r)})\}$
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

$\text{HAYCAMINO?}(\textbf{in } r\colon \texttt{red}, \textbf{in } c\colon \texttt{compu}, \textbf{in } c'\colon \texttt{compu}) \to res : \texttt{bool}$
Pre $\equiv \{(c \in \text{computadoras}(r)) \land (c' \in \text{computadoras(r)})\}$
Post $\equiv \{res =_{\text{obs}} \text{hayCamino?}(r,\ c,\ i)\}$
Complejidad: $O(L)$
Descripción: Indica si existe algún camino entre c y c'

$\text{COPIAR}(\textbf{in } r\colon \texttt{red}) \to res : \texttt{red}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} r\}$
Complejidad: $O(n! * (n^6))$
Descripción: Devuelve una copia la red

$\bullet = \bullet(\textbf{in } r\colon \texttt{red}, \textbf{in } r'\colon \texttt{red}) \to res : \texttt{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} (r =_{\text{obs}} r')\}$
Complejidad: $O(n + L^2)$
Descripción: Indica si r es igual a r'

2.2. Representación

2.2.1. Estructura

`red` se representa con `estr`

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

donde `nodoRed` es `tupla`(*pc*: `puntero(compu)` ,
caminos: `diccString(conj(lista(compu)))` ,
conexiones: $\texttt{dicc}_{Lineal}$`(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 $\Longleftrightarrow$ (

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

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

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

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

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

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

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

)

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) $\equiv$ **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 | e.compus $=_{\text{obs}}$ 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)
))))

2.3. Algoritmos

iIniciarRed () $\rightarrow$ 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*L)
```
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(L)
    while HaySiguiente?(itCompus) do                               O(1)
      Definir(cams, Siguiente(itCompus).ip, Vacio())               O(L)
      Avanzar(itCompus)                                            O(1)
    end while                                                      O(n*L)
```
Complejidad : $O(n * L)$

iConectar (**in/out** r : `red`, **in** c_0 : `compu`, **in** c_1 : `compu`, **in** i_0 : `compu`, **in** i_1 : `compu`)

```
    nr0:nodoRed ← Significado(r.dns, c0.ip)                        O(L)
    nr1:nodoRed ← Significado(r.dns, c1.ip)                        O(L)
    DefinirRapido(nr0.conexiones, i0, nr1)                         O(1)
    DefinirRapido(nr1.conexiones, i1, nr0)                         O(1)
    CrearTodosLosCaminos(r)                                        O(n! * (n^3 *(n + L)))
```
Complejidad : $O(n! * (n^3 * (n + L)))$

CrearTodosLosCaminos (**in/out** r : `red`)

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

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

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

      Avanzar(itCompuB)                                            O(1)
```

```
    end while                                                   O(n! * (n^2 *(n + L)))

    Avanzar(itCompuA)                                           O(1)
  end while                                                     O(n! * (n^3 *(n + L)))
```

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

Caminos (**in** $c1$: `nodoRed`, **in** $ipDestino$: `string`) → res: conj(lista(compu))

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

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

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

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

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

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

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

    pCandidatos ← Primero(iCandidatos)                          O(1)

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

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

        if ¬nodoEnLista(pCandidatos, iRecorrido) then           O(n*(n + L))
          iRecorrido ← Copiar(iRecorrido)                       O(n)
          AgregarAtras(iRecorrido, ∗(pCandidatos))              O(n)
          Apilar(frameRecorrido, iRecorrido)                    O(1)
          Apilar(frameCandidatos, listaNodosVecinos(pCandidatos))  O(n)
        fi                                                      O(n*(n + L))
      fi                                                        O(n*(n + L))

    fi                                                          O(n*(n + L))

  end while                                                     O(n! * n*(n + L))
```

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

Minimos (**in** $caminos$: `conj(lista(compu))`) → res: conj(lista(compu))

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

listaNodosVecinos (**in** n : `nodoRed`) → res: lista(nodoRed)

```
  res ← Vacia()                                                            O(1)
  itVecinos :itDicc(interfaz, puntero(nodoRed))) ← CrearIt(n,conexiones)   O(1)
  while HaySiguiente?(itVecinos) do                                        O(1)
    AgregarAdelante(res, *SiguienteSignificado(itVecinos))                 O(1)
    Avanzar(itVecinos)                                                     O(1)
  end while                                                                O(n)
```
Complejidad : $O(n)$

nodoEnLista (**in** n : `nodoRed`, **in** ns : `lista(nodoRed)`) → res: bool

```
  res ← false                                        O(1)
  itNodos: itLista(lista(nodoRed)) ← CrearIt(ns)     O(1)
  while HaySiguiente?(itNodos) do                    O(1)
    if Siguiente(itNodos) = n then                   O(n + L)
      res ← true                                     O(1)
    end if                                           O(1)
    Avanzar(itNodos)                                 O(1)
  end while                                          O(n*(n + L))
```
Complejidad : $O(n * (n + L))$

iComputadoras (**in** r : `red`) → res: conj(compu)

```
    res  ← r.compus                                  O(1)
```
Complejidad : $O(1)$

iConectadas? (**in** r : `red`, **in** c_0 : `compu`, **in** c_1 : `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** c_0 : `compu`, **in** c_1 : `compu`) → res: interfaz

```
  nr0:nodoRed ← Significado(r.dns, c0.ip)                        O(L)
  it :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(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)
    Avanzar(itCompus)                                            O(1)
```

```
  end while                                                        O(L*(n^2))
  itCompus CrearIt(r.compus)                                       O(1)
  while HaySiguiente?(itCompus) do                                 O(1)
    nr:nodoRed ← Significado(r.dns, Siguiente(itCompus).ip)        O(L)
    itVecinos :itDicc(interfaz, puntero(nodoRed))) ← CrearIt(conex) O(1)
    while HaySiguiente?(itVecinos) do                              O(1)
      iConectar(res, Siguiente(itCompus), ∗SiguienteSignificado
        (itVecinos))                                               O(n!*(n^4))
    end while                                                      O(n!*(n^5))
    Avanzar(itCompus)                                              O(1)
  end while                                                        O(n!*(n^6))
```

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

• = • (**in** r_0 : `red`, **in** r_1 : `red`) → res: bool

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

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

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

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

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) $\vee_L$ proximo(c) $> e$ **then** e **else** próximo(c) **fi**

desencolar(encolar(e, c)) $\equiv$ **if** vacía?(c) $\vee_L$ 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 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) $\wedge$ $c =_{\text{obs}} c_0$}
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(`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

$\bullet = \bullet$(**in** c: `colaMinPrior(`α`)`, **in** c': `colaMinPrior(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ ($c =_{\text{obs}} c'$)}
Complejidad: O(n^2)
Descripción: Indica si c es igual c'

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 $\Longleftrightarrow$
$(\forall n : \text{nat})\ \text{def?}(n, e) \Rightarrow_{\text{L}} ((\text{obtener}(n, e).\text{prioridad} = n) \wedge$
$\neg\text{vacía?}(\text{obtener}(n, e).\text{encolados}))$

3.3.3. Función de Abstracción

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

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

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

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

```
    res ← c0 = c1
```

Complejidad : $O(n^2)$

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

BORRAR(**in** c: `string`, **in/out** d: $\texttt{dicc}_{avl}(\alpha)$)
Pre $\equiv$ {def?(c, d)}
Post $\equiv$ {$res =_{\text{obs}}$ borrar(c, d))}
Complejidad: $O(log(\#claves(d)))$
Descripción: Borra el elemento con la clave dada

#CLAVES(**in** d: $\texttt{dicc}_{avl}(\alpha)$) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ #claves(d)}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de claves en el diccionario

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 × α) d ⟶ nat {#claves(d) >0}

darClaveMínima : dicc(nat × α) d × conj(nat) c ⟶ nat {(#claves(d) >0) ∧ (c ⊆ claves(d))}

claveMínima(d) ≡ darClaveMínima(d, claves(d))

darClaveMínima(d, c) ≡ **if** ∅?(sinUno(c)) **then**
dameUno(c)
else
min(dameUno(c), darClaveMínima(d, sinUno(c)))
fi

4.2. Representación

4.2.1. Representación de dicc$_{avl}$(α)

`dicc`$_{avl}$(α) 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* ∈ {-1, 0, 1 } donde *balance* es:
* 0 si el arbol esta balanceado
* 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) Todas las claves son distintas.

Rep : estr ⟶ bool

Rep(e) ≡ true ⟺ esABB(e) ∧ balanceadoBien(e) ∧ clavesDistintas(e, vacío)

esABB : puntero(nodoAvl) ⟶ bool

balanceadoBien : puntero(nodoAvl) ⟶ bool

clavesDistintas : puntero(nodoAvl) × conj(nat) ⟶ bool

balanceado : puntero(nodoAvl) ⟶ bool

esABB(n) ≡ (n ≠ NULL) $\Rightarrow_L$ (
((prim(n→hijos) ≠ NULL) $\Rightarrow_L$ (n→clave > prim(n.hijos) ∧ esABB(prim(n→hijos)))) ∧
((ult(n→hijos) ≠ NULL) $\Rightarrow_L$ (n→clave < ult(n→hijos) ∧ esABB(ult(n→hijos)))))

balanceadoBien(n) ≡ (balanceado(n) $\wedge_L$ (n ≠ NULL) $\Rightarrow_L$ (
if ((prim(n→hijos) ≠ NULL) ∧ (ult(n→hijos) ≠ NULL)) **then**
balanceadoBien(prim(n→hijos)) ∧ balanceadoBien(ult(n→hijos)))
else
if (prim(n.hijos) ≠ NULL) **then**
n→balance = 1
else
if (prim(n.hijos) ≠ NULL) **then** n→balance = -1 **else** n→balance = 0 **fi**
fi
fi

$$
\begin{array}{lll}
\text{clavesDistintas}(n,\, cs) & \equiv & (n \neq \text{NULL}) \Rightarrow_{\text{L}} \\
 & & \; n \to \text{clave} \notin cs \,\wedge \\
 & & \text{clavesDistintas}(\text{prim}(n \to \text{hijos}),\, \text{Ag}(n \to \text{clave},\, cs)) \,\wedge \\
 & & \text{clavesDistintas}(\text{ult}(n \to \text{hijos}),\, \text{Ag}(n \to \text{clave},\, cs)) \\
\text{balanceado}(n) & \equiv & (n \neq \text{NULL}) \Rightarrow_{\text{L}} \\
 & & (\textbf{if } ((\text{prim}(n \to \text{hijos}) \neq \text{NULL}) \wedge (\text{ult}(n \to \text{hijos}) \neq \text{NULL})) \;\textbf{ then} \\
 & & \quad \text{balanceado}(\text{prim}(n \to \text{hijos})) \wedge \text{balanceado}(\text{ult}(n \to \text{hijos}))) \\
 & & \textbf{else} \\
 & & \quad \textbf{if } (\text{prim}(n.\text{hijos}) \neq \text{NULL}) \;\textbf{ then} \\
 & & \qquad \text{false} \\
 & & \quad \textbf{else} \\
 & & \qquad \textbf{if } (\text{prim}(n.\text{hijos}) \neq \text{NULL}) \;\textbf{ then }\; \text{false} \;\textbf{ else }\; \text{true} \;\textbf{ fi} \\
 & & \quad \textbf{fi} \\
 & & \textbf{fi} \;)
\end{array}
$$

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)

$$
\begin{array}{lll}
\text{auxAbs}(n,\, d) & \equiv & \textbf{if } n = \text{NULL} \;\textbf{ then} \\
 & & \quad d \\
 & & \textbf{else} \\
 & & \quad \text{definir}(n.\text{clave},\, n.\text{data},\, \text{auxAbs}(\text{prim}(n.\text{hijos}),\, \text{auxAbs}(\text{ult}(n.\text{hijos}),\, d))) \\
 & & \textbf{fi}
\end{array}
$$

4.3. Algoritmos

iDefinir (**in/out** $dicc_{avl}(\alpha)$: `tree`, **in** nat : `c`, **in** α : `s`)

```
if (tree = NULL) then                                          O(1)
    tree ← crearNodo(c, s)                                     O(copy(s))
else
    it: puntero(nodoAvl) ← tree                                O(1)
    up: pila(puntero(nodoAvl))                                 O(1)
    upd: pila(int)                                             O(1)

    break: bool ← false                                        O(1)
    while(break = false)                                       O(1)
        if (it → clave < c) then                               O(1)
        Apilar(upd, 1)                                         O(1)
        else
        Apilar(upd, 0)                                         O(1)
        end if
        Apilar(up, it)                                         O(1)

        if (it → hijos[Tope(upd)] = NULL)                      O(1)
            break ← true                                       O(1)
        end if

        it ← (it → hijos[Tope(upd)])                           O(1)
    do

    (it → hijos[Tope(upd)]) ← crearNodo(c, s)                  O(1)

    break ← false                                              O(1)
    while((Tamano(up) > 0) ∧ (break = false))                  O(1)
```

```
            if(Tope(upd) = 0) then                                              O(1)
                (Tope(up) → balance) ← (Tope(up) → balance) − 1                 O(1)
            else
                (Tope(up) → balance) ← (Tope(up) → balance) + 1                 O(1)
            end if

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

                    if(Tamano(up) > 1) then                                     O(1)
                        upTope: puntero(nodoAvl) ← Tope(up)                     O(1)
                        Desapilar(up)                                           O(1)
                        Desapilar(upd)                                          O(1)
                        (Tope(up) → hijos[Tope(upd)]) ← upTope                  O(1)
                    else
                        tree ← Tope(up)                                         O(1)
                    end if

                    break ← true                                                O(1)
                end if
            end if
            Desapilar(up)                                                       O(1)
            Desapilar(upd)                                                      O(1)
        do                                                                      O(log(k))
end if
```
Complejidad : $O(log(k)) + O(copy(s))$

crearNodo (**in** nat : `c`, **in** α : `s`) → res: puntero(nodoAvl)

```
    hijos: arreglo_estatico[1] de puntero(nodoAvl)                              O(1)
    hijos[0] ← NULL                                                             O(1)
    hijos[1] ← NULL                                                             O(1)
    res ← puntero(<c, copy(s), 0, hijos>)                                       O(copy(s))
```
Complejidad : $O(copy(s))$

insertarBalance (**in/out** $puntero(nodoAvl)$: `root`, **in** int : `dir`) → res: puntero(nodoAvl)

```
    nodo: puntero(nodoAvl) ← (root → hijos[dir])                                O(1)

    if(dir = 0) then                                                            O(1)
        bal: int ← −1                                                           O(1)
    else
        bal: int ← 1                                                            O(1)
    end if

    if(nodo → balance = bal) then                                               O(1)
        (root → balance) ← 0                                                    O(1)
        (nodo → balance) ← 0                                                    O(1)
        root ← rotacionSimple(root, ¬dir)                                       O(1)
    else
        ajustarBalance(root, dir, bal)                                          O(1)
        root ← rotacionDoble(root, ¬dir)                                        O(1)
    end if
```

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

rotacionSimple (**in/out** *puntero(nodoAvl)* : `root`, **in** *int* : `dir`) → res: puntero(nodoAvl)

```
    nodo: puntero(nodoAvl) ← (root → hijos[¬dir])                     O(1)

    (root → hijos[¬dir]) ← (nodo → hijos[dir])                        O(1)
    (nodo → hijos[dir]) ← root                                        O(1)

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

rotacionDoble (**in/out** *puntero(nodoAvl)* : `root`, **in** *int* : `dir`) → res: puntero(nodoAvl)

```
    nodo: puntero(nodoAvl) ← ((root → hijos[¬dir]) → hijos[dir])      O(1)

    ((root → hijos[¬dir]) → hijos[dir]) ← (nodo → hijos[¬dir])        O(1)
    (nodo → hijos[¬dir]) ← (root → hijos[¬dir])                       O(1)
    (root → hijos[¬dir]) ← nodo                                       O(1)

    nodo ← (root → hijos[¬dir])                                       O(1)
    (root → hijos[¬dir]) ← (nodo → hijos[dir])                        O(1)
    (nodo → hijos[dir]) ← root                                        O(1)

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

ajustarBalance (**in/out** *puntero(nodoAvl)* : `root`, **in** *int* : `dir`, **in** *int* : `bal`) → res: puntero(nodoAvl)

```
    nodo: puntero(nodoAvl) ← (root → hijos[dir])                      O(1)
    nodoHijo: puntero(nodoAvl) ← (nodoUno → hijos[¬dir])              O(1)

    if(nodoHijo → balance = 0) then                                   O(1)
        (root → balance) ← 0                                          O(1)
        (nodo → balance) ← 0                                          O(1)
    else
        if(nodoHijo → balance = bal) then                             O(1)
            (root → balance) ← −bal                                   O(1)
            (nodo → balance) ← 0                                      O(1)
        else
            (root → balance) ← 0                                      O(1)
            (nodo → balance) ← bal                                    O(1)
        end if
    end if

    (nodoHijo → balance)← 0                                           O(1)
```
Complejidad : $O(1)$

iBorrar (**in/out** $dicc_{avl}(\alpha)$: `tree`, **in** *nat* : `c`)

```
    if(tree != NULL) then                                             O(1)
        it: puntero(nodoAvl) ← tree                                   O(1)
        up: pila(puntero(nodoAvl))                                    O(1)
        upd: pila(int)                                                O(1)
```

```
break: bool ← false                                              O(1)
while(break = false)                                             O(1)
    if (it → clave = c) then                                     O(1)
        break ← true                                             O(1)
    end if

    if (it → clave < c) then                                     O(1)
        Apilar(upd, 1)                                           O(1)
    else
        Apilar(upd, 0)                                           O(1)
    end if
    Apilar(up, it)                                               O(1)

    it ← (it → hijos[Tope(upd)])                                 O(1)
do                                                               O(log(k))

if((it → hijos[0] = NULL) ∨ (it → hijos[0] = NULL)) then         O(1)
    if(it → hijos[0] = NULL) then                                O(1)
        dir: int ← 1                                             O(1)
    else
        dir: int ← 0                                             O(1)
    end if

    if(Tamano(up) > 1) then                                      O(1)
        (Tope(up) → hijos[Tope(upd)]) ← (it → hijos[dir])        O(1)
    else
        tree ← (it → hijos[dir])                                 O(1)
    end if
else
    heredero: puntero(nodoAvl) ← (it → hijos[1])                 O(1)

    Tope(upd) ← 1                                                O(1)
    Tope(up) ← it                                                O(1)

    while(heredero → hijos[0] != null)                           O(1)
        Apilar(upd, 0)                                           O(1)
        Apilar(up, heredero)                                     O(1)
        heredero ← (heredero → hijos[0])                         O(1)
    do                                                           O(log(k))

    (it → clave) ← (heredero → clave)                            O(1)

    Desapilar(up)                                                O(1)
    Desapilar(upd)                                               O(1)

    if(Tope(up) = it) then                                       O(1)
        (Tope(up) → hijos[1]) ← (heredero → hijos[1])            O(1)
    else
        (Tope(up) → hijos[0]) ← (heredero → hijos[1])            O(1)
    end if
end if

break ← false                                                    O(1)
while((break = false) ∧ (Tamano(up) ≥ 0))                        O(1)
    if(Tope(upd) != 0) then                                      O(1)
        (Tope(up) → balance) ← (Tope(up) → balance) − 1          O(1)
    else
        (Tope(up) → balance) ← (Tope(up) → balance) + 1          O(1)
```

```
            end if

            if(abs(Tope(up) → balance) = 1) then                                      O(1)
                break ← true                                                           O(1)
            else
                if(abs(Tope(up) → balance) > 1) then                                  O(1)
                    Tope(up) ← removerBalanceo(Tope(up), Tope(upd), \&break)          O(1)
                    if(Tamano(up) > 1) then                                            O(1)
                        upTope: puntero(nodoAvl) ← Tope(up)                            O(1)
                        Desapilar(up)                                                  O(1)
                        Desapilar(upd)                                                 O(1)
                        (Tope(up) → hijos[Tope(upd)]) ← upTope                         O(1)
                    else
                        tree ← Tope(up)                                                O(1)
                    end if
                end if
            end if
        do                                                                             O(log(k))
    end if
```
Complejidad : $O(log(k))$

removerBalanceo (**in/out** *puntero*(*nodoAvl*) : `root`, **in** *int* : `dir`, **in/out** *puntero*(*bool*) : `done`)

```
    nodo: puntero(nodoAvl) ← (root → hijos[¬dir])                                    O(1)

    if(dir = 0) then                                                                   O(1)
        bal ← −1                                                                       O(1)
    else
        bal ← 1                                                                        O(1)
    end if

    if(nodo → balance =−bal) then                                                      O(1)
        (root → balance) ← 0                                                           O(1)
        (nodo → balance) ← 0                                                           O(1)
        root ← rotacionSimple(root, dir)                                               O(1)
    else
        if((nodo → balance) = bal) then                                                O(1)
            ajustarBalance(root, ¬dir, −bal)                                           O(1)
            root ← rotacionDoble(root, dir)                                            O(1)
        else
            (root → balance) ← −bal                                                    O(1)
            (nodo → balance) ← bal                                                     O(1)
            root ← rotacionSimple(root, dir)                                           O(1)
            ∗done ← true                                                               O(1)
        end if
    end if

    res ← root                                                                         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`) → 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
```

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

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

5. Módulo Árbol binario(α)

5.1. Interfaz

se explica con: ÁRBOL BINARIO(α).

géneros: `ab`(α).

5.1.1. Operaciones básicas de Árbol binario(α)

NIL() $\rightarrow$ *res* : `ab`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ nil}
Complejidad: $O(1)$
Descripción: Crea un árbol binario nulo

BIN(**in** i: `ab`(α), **in** r: α, **in** d: `ab`(α)) $\rightarrow$ *res* : `ab`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ bin(i, r, d)}
Complejidad: $O(copy(r) + copy(i) + copy(d))$
Descripción: Crea un árbol binario con hijo izquierdo i, hijo derecho d y raíz de valor r

RAÍZ(**in/out** a: `ab`(α)) $\rightarrow$ *res* : α
Pre $\equiv$ {$\neg$nil?(a)}
Post $\equiv$ {alias($res =_{\text{obs}}$ raíz(a))}
Complejidad: $O(1)$
Descripción: Devuelve el valor de la raíz del árbol
Aliasing: *res* es modificable si y sólo si a lo es

IZQ(**in/out** a: `ab`(α)) $\rightarrow$ *res* : `ab`(α)
Pre $\equiv$ {$\neg$nil?(a)}
Post $\equiv$ {alias($res =_{\text{obs}}$ izq(a))}
Complejidad: $O(1)$
Descripción: Devuelve el hijo izquierdo
Aliasing: *res* es modificable si y sólo si a lo es

DER(**in/out** a: `ab`(α)) $\rightarrow$ *res* : `ab`(α)
Pre $\equiv$ {$\neg$nil?(a)}
Post $\equiv$ {alias($res =_{\text{obs}}$ der(a))}
Complejidad: $O(1)$
Descripción: Devuelve el hijo derecho
Aliasing: *res* es modificable si y sólo si a lo es

NIL?(**in/out** a: `ab`(α)) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ nil?(a)}
Complejidad: $O(1)$
Descripción: Devuelve `true` si *res* es un árbol vacío

5.2. Representación

5.2.1. Representación de ab(α)

`ab(`α`)` se representa con `estr`

donde `estr` es `puntero(nodoAb)`

donde `nodoAb` es `tupla(` *raiz*: α, *hijos*: `arreglo[2] de ab(`α`)`)

5.2.2. Invariante de Representación

(I) No puede haber ciclos en el árbol

(II) Los hijos no pueden apuntar a un mismo árbol

5.2.3. Función de Abstracción

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

Abs(e) $=_{\text{obs}}$ abn: ab(α) | (nil?(abn) $\Leftrightarrow$ e = NULL) $\wedge$
(¬nil?(abn) $\Rightarrow_{\text{L}}$
(raíz(abn) = $e \rightarrow$ raíz $\wedge$ izq(abn) = $e \rightarrow$ hijos[0] $\wedge$ der(abn) = $e \rightarrow$ hijos[1])
)

5.3. Algoritmos

iNil () → res: ab(α)

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

Complejidad : $O(1)$

iBin (**in** i : `ab(`α`)`, **in** r : α, **in** d : `ab(`α`)`) → res: ab(α)

```
nuevoAb:nodoAb                                      O(1)
nuevoAb.raiz ← copy(r)                              O(copy(r))
nuevoAb.hijos[0] ← copy(i)                          O(copy(i))
nuevoAb.hijos[1] ← copy(d)                          O(copy(d))

res ← puntero(nuevoAb)                              O(1)
```

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

iRaíz (**in/out** a : `ab(`α`)`) → res: α

```
res ← (a → raiz)                                    O(1)
```

Complejidad : $O(1)$

iIzq (**in/out** a : `ab(`α`)`) → res: ab(α)

res ← (a → hijos[0]) O(1)

Complejidad : $O(1)$

iDer (**in/out** a : `ab(`α`)`) → res: ab(α)

res ← (a → hijos[1]) O(1)

Complejidad : $O(1)$

iNil? (**in** a : `ab(`α`)`) → res: bool

res ← (a = NULL) O(1)

Complejidad : $O(1)$

6. Módulo Diccionario Trie(α)

6.1. Interfaz

se explica con: DICCIONARIO(STRING, α). **géneros**: `diccString`(α).

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

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

DEFINIDO?(**in** c: `string`, **in** d: `diccString`(α))) $\rightarrow$ *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: α, **in/out** d: `diccString`(α))
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: `diccString`(α)) $\rightarrow$ *res* : α
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.

$\bullet = \bullet$(**in/out** d: `diccString`(α), **in/out** d': `diccString`(α)) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}}$ (d $=_{\text{obs}}$ d')$\}$
Complejidad: $O(L * n * (\alpha =_{\text{obs}} \alpha'))$
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.

COPIAR(**in** *dicc*: `diccString`(α)) $\rightarrow$ *res* : `diccString`(α)
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}}$ dicc$\}$
Complejidad: $O(n * L)$
Descripción: Devuelve una copia del diccionario