

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** $r : \text{red}$) $\rightarrow res : \text{dcnet}$

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

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

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** $dcn : \text{dcnet}$, **in** $p : \text{paquete}$)

Pre $\equiv \{$

$dcn =_{\text{obs}} dcn_0 \wedge$

$\neg((\exists p' : \text{paquete})(\text{paqueteEnTransito}(dcn, p') \wedge \text{id}(p) = \text{id}(p') \wedge \text{origen}(p) \in \text{computadoras}(\text{red}(dcn)) \wedge_L$

$\text{destino}(p) \in \text{computadoras}(\text{red}(dcn)) \wedge_L \text{hayCamino}?(\text{red}(dcn), \text{origen}(p), \text{destino}(p))))$

$\}$

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

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** $dcn : \text{dcnet}$)

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

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

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** $dcn : \text{dcnet}$) $\rightarrow res : \text{red}$

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

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

Complejidad: $O(1)$

Descripción: devuelve la red de una DCNet

Aliasing: res es una referencia no modificable

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

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

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

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

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

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

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

Descripción: devuelve la cantidad de paquetes enviados por una compu

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

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

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

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

Descripción: devuelve el conjunto de paquetes encolados en una compu

Aliasing: res es una referencia no modificable

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

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

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

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

Descripción: indica si el paquete está en transito

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

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

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

Complejidad: $O(1)$

Descripción: devuelve la compu que mas paquetes envió

Aliasing: res es una referencia no modificable

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

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

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

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

Descripción: compara dcn_1 y dcn_2 por igualdad

1.2. Representación

1.2.1. Representación de dcnet

dcnet se representa con estr

donde estr es $\text{tupla}(\text{topología: red,}$
 $\text{vectorCompusDCNet: vector(compuDCNet),}$
 $\text{diccCompusDCNet: dicc}_{\text{trie}}(\text{puntero(compuDCNet)}),$
 $\text{conjPaquetesDCNet: conj(paqueteDCNet),}$
 $\text{laQueMásEnvio: puntero(compuDCNet)})$

donde compuDCNet es $\text{tupla}(pc: \text{puntero(compu),}$
 $\text{conjPaquetes: conj(paquete),}$
 $\text{diccPaquetesDCNet: dicc}_{\text{avl}}(\text{nat, itConj(paqueteDCNet)}),$
 $\text{colaPaquetesDCNet: colaPrioridad(nat, itConj(paqueteDCNet))},$
 $\text{paqueteAEnviar: itConj(paqueteDCNet), enviados: nat})$

donde paqueteDCNet es $\text{tupla}(it: \text{itConj(paquete), recorrido: lista(compu)})$

donde paquete es $\text{tupla}(id: \text{nat, prioridad: nat, origen: compu, destino: compu})$

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

1.2.2. Invariante de Representación

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

Rep : estr \rightarrow bool

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

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

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

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

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

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

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

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

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

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

```

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

```


1.2.3. Función de Abstracción

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

1.3. Algoritmos

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

```

res.topologia  $\leftarrow$  Copiar(topo) O(n! * n^6)
res.vectorCompusDCNet  $\leftarrow$  Vacía() O(1)
res.diccCompusDCNet  $\leftarrow$  CrearDicc() O(1)
res.laQueMasEnvio  $\leftarrow$  NULL O(1)
res.conjPaquetesDCNet  $\leftarrow$  Vacío() O(1)

itConj(compu): it  $\leftarrow$  CrearIt(Computadoras(topo)) O(1)

if (HaySiguierte?(it)) then O(1)
    res.laQueMasEnvio  $\leftarrow$  puntero(Siguierte(it)) O(1)
end if

while HaySiguierte?(it) do O(1)
    compuDCNet: computdcnet  $\leftarrow$  <puntero(Siguierte(it)), Vacío(), CrearDicc(),
        Vacía(), CrearIt(Vacío()), 0> O(1)
    AgregarAtras(res.vectorCompusDCNet, computdcnet) O(n)
    Definir(res.diccCompusDCNet, Siguierte(it).ip, puntero(computdcnet)) 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): computdcnet  $\leftarrow$ 
    Significado(dcn.diccCompusDCNet, p.origen.ip) O(L)
itConj(paquete): itPaq  $\leftarrow$  AgregarRapido(computdcnet  $\rightarrow$  conjPaquetes, p) O(1)
lista(compu): recorr  $\leftarrow$  AgregarAtras(Vacía(), p.origen) O(1)
paqueteDCNet: paqDCNet  $\leftarrow$  <itPaq, recorr> O(1)

itConj(paqueteDCNet): itPaqDCNet  $\leftarrow$ 
    AgregarRapido(dcn.conjPaquetesDCNet, paqDCNet) O(1)
Definir(computdcnet  $\rightarrow$  diccPaquetesDCNet, itPaqDCNet) O(\log(k))
Encolar(computdcnet  $\rightarrow$  colaPaquetesDCNet, itPaqDCNet) O(\log(k))

```

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

iAvanzarSegundo (in/out dcn: dcnet)

```

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

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

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

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

        if (pAEnviar.destino ≠ siguientecompu) then

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

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

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

            AgregarAtras(Siguiente(paqAEnviar).recorrido, siguientecompu)

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

        end if

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

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

    end if
    i++
end while

```

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

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

res \leftarrow dcn.topologia

$O(1)$

Complejidad : $O(1)$

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

nat: i \leftarrow 0

$O(1)$

while i < Longitud(dcn.vectorCompusDCNet) do

$O(1)$

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

$O(\log(k))$

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

$O(\log(k))$

end if

i++

$O(1)$

end while

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

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

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

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

$O(L)$

Complejidad : $O(L)$

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

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

$O(L)$

Complejidad : $O(L)$

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

res \leftarrow false

nat: i \leftarrow 0

$O(1)$

while i < Longitud(dcn.vectorCompusDCNet) do

$O(1)$

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

$O(\log(k))$

res \leftarrow true

$O(1)$

end if

i++

$O(1)$

end while

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

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

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

```
res ← *(dcn.laQueMasEnvio→pc)
```

O(1)

Complejidad : $O(1)$

• =_i • (in dcn_1 : dcnnet, in dcn_2 : dcnnet) → 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 ← Siguiete(c1.paqueteAEnviar) =paqdcn Siguiete(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 HaySiguiete?(itconj1) do
```

O(1)

```
    if Definido?(c2.diccPaquetesDCNet, Siguiete(itconj1)).id then
```

O(log(n))

```
      if ¬(Siguiete(Obtener(c1.diccPaquetesDCNet, Siguiete(itconj1)).id))
```

=_{paqdcn}

```
        Siguiete(Obtener(c1.diccPaquetesDCNet, Siguiete(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 ¬(Siguiete(Proximo(c1.colaPrioridad)) =paqdcn
```

```
    Siguiete(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( $p_1.it$ ) = Siguiente( $p_2.it$ )      O(1)

```

```

bool: boolRecorrido ←  $p_1.recorrido$  =  $p_2.recorrido$       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 $\equiv \{res =_{obs} iniciarRed\}$
Complejidad: $O(1)$
Descripción: Crea una red nueva

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

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

COMPUTADORAS(**in** $r : red$) $\rightarrow res : conj(compu)$
Pre $\equiv \{true\}$
Post $\equiv \{alias(res =_{obs} computadoras(r))\}$
Complejidad: $O(1)$
Descripción: Devuelve las computadoras de la red [Devuelve una referencia no modificable]

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

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

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

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

Complejidad: $O(L + n)$

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

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

Complejidad: $O(L)$

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

Aliasing: Devuelve una referencia no modificable

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

Complejidad: $O(L)$

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

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

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

Descripción: Devuelve una copia la red

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

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

Descripción: Indica si r es igual a r'

2.2. Representación

2.2.1. Estructura

red se representa con estr

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

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

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

2.2.2. Invariante de Representación

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

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

Rep : estr \rightarrow bool

Rep(e) \equiv true \iff (

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

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

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

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

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

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

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

)

| | | |
|-------------------|--|--|
| vecinas | : nodoRed | \rightarrow conj(nodoRed) |
| auxVecinas | : nodoRed \times dicc(nat \times puntero(nodoRed)) | \rightarrow conj(nodoRed) |
| secusDeLongK | : conj(secu(α)) \times nat | \rightarrow conj(secu(α)) |
| longMenorSec | : conj(secu(α)) secu | \rightarrow nat $\{ \neg \emptyset?(secus) \}$ |
| darRutas | : nodoRed $nA \times$ nodoRed $nB \times$ conj(pc) \times secu(nodoRed) | \rightarrow conj(secu(nodoRed)) |
| darRutasVecinas | : conj(pc) vec \times nodoRed $n \times$ conj(pc) \times secu(nodoRed) | \rightarrow conj(secu(nodoRed)) |
| darCaminosMinimos | : nodoRed $n1 \times$ nodoRed $n1$ | \rightarrow conj(secu(compu)) |

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

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

```

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

```

2.2.3. Función de Abstracción

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

Abs(*e*) =_{obs} *r*: red | *e*.compus =_{obs} computadoras(*r*) \wedge

(($\forall c1, c2: \text{compu}, i1, i2: \text{string}, n1, n2: \text{nodoRed}$) (

($c1 \in e.\text{compus} \wedge i1 = c1.\text{ip} \wedge \text{def?}(i1, e.\text{dns}) \wedge_L n1 = \text{obtener}(i1, e.\text{dns}) \wedge c1 = *n1.\text{pc}$) \wedge

($c2 \in e.\text{compus} \wedge i2 = c2.\text{ip} \wedge \text{def?}(i2, e.\text{dns}) \wedge_L n2 = \text{obtener}(i2, e.\text{dns}) \wedge c2 = *n2.\text{pc}$) \wedge

($c1 \neq c2$)) \Rightarrow_L

(conectadas?*r*, *c1*, *c2*) \Leftrightarrow ($\exists t1, t2: \text{nat}$) (

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  $\leftarrow$  Vacio()

```

O(1)

| | |
|--|--------|
| <pre>res.dns ← Vacio()</pre> | $O(1)$ |
| Complejidad : $O(1)$ | |

| | |
|--|--|
| <pre>iAgregarComputadora (in/out r: red, in c: compu) AgregoCompuNuevaAlResto(r.dns, c) AgregarRapido(r.compus, c) Definir(r.dns, compu.ip, Tupla(x, Vacio(), Vacio())>) InicializarConjCaminos(r, c)</pre> | $O(n * L)$ $O(1)$ $O(L)$ $O(n * L)$ |
| Complejidad : $O(n * L)$ | |

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

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

| | |
|--|---|
| <pre>iConectar (in/out r: red, in c0: compu, in c1: compu, in i0: compu, in i1: compu) nr0:nodoRed ← Significado(r.dns, c0.ip) nr1:nodoRed ← Significado(r.dns, c1.ip) DefinirRapido(nr0.conexiones, i0, nr1) DefinirRapido(nr1.conexiones, i1, nr0) CrearTodosLosCaminos(r)</pre> | $O(L)$ $O(L)$ $O(1)$ $O(1)$ $O(n! * (n^3 * (n + L)))$ |
| Complejidad : $O(n! * (n^3 * (n + L)))$ | |

| | |
|--|---|
| <pre>CrearTodosLosCaminos (in/out r: red) itCompuA: itConj(compu) ← CrearIt(r.compus) while HaySiguiente?(itCompuA) do nr:nodoRed ← Significado(r.dns, Siguiente(itCompuA).ip) itCompuB: itConj(compu) ← CrearIt(r.compus) while HaySiguiente?(itCompuB) do caminimos: conj(lista(compu)) ← Minimos(Caminos (nr, Siguiente(itCompuB).ip) Definir(nr.caminos, Siguiente(itCompuB).ip, caminimos) Avanzar(itCompuB)</pre> | $O(1)$ $O(1)$ $O(L)$ $O(1)$ $O(1)$ $O(n! * n * (n + L))$ $O(L)$ $O(1)$ |
|--|---|

| | |
|--|---------------------------|
| 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 ¬EsVacía?(frameRecorrido) do | $O(1)$ |
| iRecorrido ← Tope(frameRecorrido) | $O(1)$ |
| iCandidatos ← Tope(frameCandidatos) | $O(1)$ |
| Desapilar(frameRecorrido) | $O(1)$ |
| Desapilar(frameCandidatos) | $O(1)$ |
| pCandidatos ← Primero(iCandidatos) | $O(1)$ |
| if ¬EsVacío?(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
      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 c0: compu, in c1: compu) → res: bool
nr0: nodoRed ← Significado(r.dns, c0.ip)       O(L)
it : itDicc(interfaz, puntero(nodoRed)) ← CrearIt(nr0.conexiones) O(1)
res ← false                                   O(1)
while HaySiguiente?(it) do                    O(1)
  if c1.ip = SiguienteSignificado(it) → pc → ip then
    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) |
| res.compus ← Copiar(r.compus) | O(n) |
| res.dns ← Copiar(r.dns) | O(n(L + Copiar(nodoRed))) |
| Complejidad : $O(n(L + n(L + n^2)))$ | |

$$O(\text{Copiar}(\text{nodoRed})) = O(\text{copy}(\text{pc}) + \text{Copiar}(\text{conexiones}) + \text{Copiar}(\text{caminos}))$$

$$O(\text{copy}(\text{pc})) = O(1)$$

$$O(\text{Copiar}(\text{conexiones})) = O(n)$$

$$O(\text{Copiar}(\text{caminos})) = O(n(L + \text{Copiar}(\text{conj}(\text{lista}(\text{compu}))))))$$

$$O(\text{Copiar}(\text{conj}(\text{lista}(\text{compu})))) = O(n^2)$$

$$O(\text{Copiar}(\text{nodoRed})) = O(n(L + n^2))$$

$$\bullet = \bullet \text{ (in } r_0 : \text{red, in } r_1 : \text{red}) \rightarrow \text{res: bool}$$

$$\text{res} \leftarrow (r_0.\text{compus} = r_1.\text{compus}) \wedge (r_0.\text{dns} = r_1.\text{dns})$$

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

$$O(n + L^2)$$

3. Módulo Cola de mínima prioridad(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

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

parámetros formales

géneros α

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

Relación de orden total estricto¹

\bullet

géneros $\text{colaMinPrior}(\alpha)$

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

usa **BOOL**

observadores básicos

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

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

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

generadores

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

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

otras operaciones

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

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

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

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

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

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

Fin TAD

¹Una 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() $\rightarrow res : \text{colaMinPrior}(\alpha)$

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

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

Complejidad: $O(1)$

Descripción: Crea una cola de prioridad vacía

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

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

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

Complejidad: $O(1)$

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

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

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

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

Complejidad: $O(\log(\text{tamaño}(c)))$

Descripción: Quita el elemento más prioritario

Aliasing: Se devuelve el elemento por copia

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

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

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

Complejidad: $O(\log(\text{tamaño}(c)))$

Descripción: Agrega al elemento α con prioridad p a la cola

Aliasing: Se agrega el elemento por copia

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

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

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

Complejidad: $O(n^2)$

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 $\text{dicc}_{\text{avl}}(\text{nat}, \text{nodoEncolados})$

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

3.3.2. Invariante de Representación

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

Rep : estr \longrightarrow bool

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

3.3.3. Función de Abstracción

Abs : estr $e \longrightarrow$ colaMinPrior

{Rep(e)}

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

3.4. Algoritmos

iVacía () \rightarrow res: colaMinPrior(α)

res \leftarrow Vacío()

O(1)

Complejidad : O(1)

iVacía? (in c : colaMinPrior(α)) \rightarrow res: bool

res \leftarrow (#Claves(c) = 0)

O(1)

Complejidad : O(1)

iDesencolar (in/out c : colaMinPrior(α)) \rightarrow res: α

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

O(copy(α))

Desencolar(Minimo(c).encolados)

O(log(tamaño(c)))

if EsVacía?(Minimo(c).encolados) then

O(1)

 Borrar(c , Minimo(c).prioridad)

O(log(tamaño(c)))

end if

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

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

if Definido?(p) then

O(log(tamaño(c)))

 Encolar(Significado(c , p).encolados, a)

O(log(tamaño(c)) + copy(α))

else

 nodoEncolados nuevoNodoEncolados

O(1)

 nuevoNodoEncolados.encolados \leftarrow Vacía()

O(1)

 nuevoNodoEncolados.prioridad \leftarrow p

O(1)

```
    Encolar(nuevoNodoEncolados.encolados, a)                                O(copy(a))  
    Definir(c, p, nuevoNodoEncolados)                                O(log(tamaño(c)) + copy(nodoEncolados))  
end if
```

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

```
• = • (in c0: colaMinPrior( $\alpha$ ), in c1: colaMinPrior( $\alpha$ )) → res: bool  
    res ← c0 = c1
```

Complejidad : $O(n^2)$

4. Módulo Diccionario AVL(α)

4.1. Interfaz

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

géneros: $\text{dicc}_{avl}(\alpha)$.

4.1.1. Operaciones básicas de Diccionario AVL(α)

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

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

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

Complejidad: $O(1)$

Descripción: Crea un diccionario vacío

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Descripción: Borra el elemento con la clave dada

$\#CLAVES(\text{in } d : \text{dicc}_{avl}(\alpha)) \rightarrow res : \text{nat}$

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

Post $\equiv \{res =_{\text{obs}} \#claves(d)\}$

Complejidad: $O(1)$

Descripción: Devuelve la cantidad de claves en el diccionario

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

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

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

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

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

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

4.1.2. Operaciones auxiliares del TAD

$\text{claveMínima} : \text{dicc}(\text{nat} \times \alpha) \, d \longrightarrow \text{nat} \quad \{\# \text{claves}(d) > 0\}$
 $\text{darClaveMínima} : \text{dicc}(\text{nat} \times \alpha) \, d \times \text{conj}(\text{nat}) \, c \longrightarrow \text{nat} \quad \{(\# \text{claves}(d) > 0) \wedge (c \subseteq \text{claves}(d))\}$
 $\text{claveMínima}(d) \equiv \text{darClaveMínima}(d, \text{claves}(d))$
 $\text{darClaveMínima}(d, c) \equiv \text{if } \emptyset?(\text{sinUno}(c)) \text{ then}$
 $\quad \text{dameUno}(c)$
 $\quad \text{else}$
 $\quad \text{min}(\text{dameUno}(c), \text{darClaveMínima}(d, \text{sinUno}(c)))$
 $\quad \text{fi}$

4.2. Representación

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

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

donde **estr** es **puntero(nodoAvl)**

donde **nodoAvl** es **tupla(clave: nat, data: 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 $\text{balance} \in \{-1, 0, 1\}$ donde balance es:
 - * 0 si el árbol está balanceado
 - * 1 si existe un nodo en el último nivel de balance tal que tenga un hijo a la izq
 - * -1 si existe un nodo en el último nivel de balance tal que tenga un hijo a la der
- (III) Todas las claves son distintas.

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

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

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

$\text{balanceadoBien} : \text{puntero}(\text{nodoAvl}) \longrightarrow \text{bool}$

$\text{clavesDistintas} : \text{puntero}(\text{nodoAvl}) \times \text{conj}(\text{nat}) \longrightarrow \text{bool}$

$\text{balanceado} : \text{puntero}(\text{nodoAvl}) \longrightarrow \text{bool}$

$\text{esABB}(n) \equiv (n \neq \text{NULL}) \Rightarrow_L ($
 $\quad ((\text{prim}(n \rightarrow \text{hijos}) \neq \text{NULL}) \Rightarrow_L (n \rightarrow \text{clave} > \text{prim}(n \rightarrow \text{hijos}) \wedge \text{esABB}(\text{prim}(n \rightarrow \text{hijos})))) \wedge$
 $\quad ((\text{ult}(n \rightarrow \text{hijos}) \neq \text{NULL}) \Rightarrow_L (n \rightarrow \text{clave} < \text{ult}(n \rightarrow \text{hijos}) \wedge \text{esABB}(\text{ult}(n \rightarrow \text{hijos}))))))$
 $\text{balanceadoBien}(n) \equiv (\text{balanceado}(n) \wedge_L (n \neq \text{NULL}) \Rightarrow_L ($
 $\quad \text{if } ((\text{prim}(n \rightarrow \text{hijos}) \neq \text{NULL}) \wedge (\text{ult}(n \rightarrow \text{hijos}) \neq \text{NULL})) \text{ then}$
 $\quad \quad \text{balanceadoBien}(\text{prim}(n \rightarrow \text{hijos})) \wedge \text{balanceadoBien}(\text{ult}(n \rightarrow \text{hijos}))$
 $\quad \text{else}$
 $\quad \quad \text{if } (\text{prim}(n \rightarrow \text{hijos}) \neq \text{NULL}) \text{ then}$
 $\quad \quad \quad n \rightarrow \text{balance} = 1$
 $\quad \quad \text{else}$
 $\quad \quad \quad \text{if } (\text{prim}(n \rightarrow \text{hijos}) \neq \text{NULL}) \text{ then } n \rightarrow \text{balance} = -1 \text{ else } n \rightarrow \text{balance} = 0 \text{ fi}$
 $\quad \text{fi}$
 $\quad \text{fi}$

```

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

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

```

4.2.3. Función de Abstracción

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

```

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

```

4.3. Algoritmos

```

iDefinir (in/out  $\text{dicc}_{\text{avl}}(\alpha) : \text{tree}$ , in  $\text{nat} : c$ , in  $\alpha : s$ )
    if ( $\text{tree} = \text{NULL}$ ) then O(1)
         $\text{tree} \leftarrow \text{crearNodo}(c, s)$  O(\text{copy}(s))
    else
        it: puntero(nodoAvl)  $\leftarrow \text{tree}$  O(1)
        up: pila(puntero(nodoAvl)) O(1)
        upd: pila(int) O(1)

        break: bool  $\leftarrow \text{false}$  O(1)
        while(break = false) O(1)
            if (it  $\rightarrow \text{clave} < c$ ) then O(1)
                Apilar(upd, 1) O(1)
            else
                Apilar(upd, 0) O(1)
            end if
            Apilar(up, it) O(1)

            if (it  $\rightarrow \text{hijos}[\text{Tope}(\text{upd})] = \text{NULL}$ ) O(1)
                break  $\leftarrow \text{true}$  O(1)
            end if

            it  $\leftarrow (\text{it} \rightarrow \text{hijos}[\text{Tope}(\text{upd})])$  O(1)
        do

        ( $\text{it} \rightarrow \text{hijos}[\text{Tope}(\text{upd})]) \leftarrow \text{crearNodo}(c, s)$  O(1)

        break  $\leftarrow \text{false}$  O(1)
        while(( $\text{Tamano}(\text{up}) > 0$ )  $\wedge$  (break = false)) O(1)

```

```

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

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

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

        break ← true
    end if
end if
Desapilar(up)
Desapilar(upd)
do
end if
Complejidad :  $O(\log(k)) + O(\text{copy}(s))$ 

```

```

crearNodo (in nat: c, in α: s) → res: puntero(nodoAvl)
    hijos: arreglo_estatico[1] de puntero(nodoAvl)
    hijos[0] ← NULL
    hijos[1] ← NULL
    res ← puntero(<c, copy(s), 0, hijos>)
Complejidad :  $O(\text{copy}(s))$ 

```

```

insertarBalance (in/out puntero(nodoAvl): root, in int: dir) → res: puntero(nodoAvl)
    nodo: puntero(nodoAvl) ← (root → hijos[dir])

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

    if (nodo → balance = bal) then
        (root → balance) ← 0
        (nodo → balance) ← 0
        root ← rotacionSimple(root, -dir)
    else
        ajustarBalance(root, dir, bal)
        root ← rotacionDoble(root, -dir)
    end if

```

| | |
|--|------|
| <pre>res ← root</pre> | O(1) |
| Complejidad : $O(1)$ | |

| | |
|--|------|
| <pre>rotacionSimple (in/out puntero(nodoAvl): root, in int: dir) → res: puntero(nodoAvl) nodo: puntero(nodoAvl) ← (root → hijos[¬dir])</pre> | O(1) |
| <pre> (root → hijos[¬dir]) ← (nodo → hijos[dir])</pre> | O(1) |
| <pre> (nodo → hijos[dir]) ← root</pre> | O(1) |
| <pre> res ← nodo</pre> | O(1) |
| Complejidad : $O(1)$ | |

| | |
|--|------|
| <pre>rotacionDoble (in/out puntero(nodoAvl): root, in int: dir) → res: puntero(nodoAvl) nodo: puntero(nodoAvl) ← ((root → hijos[¬dir]) → hijos[dir])</pre> | O(1) |
| <pre> ((root → hijos[¬dir]) → hijos[dir]) ← (nodo → hijos[¬dir])</pre> | O(1) |
| <pre> (nodo → hijos[¬dir]) ← (root → hijos[¬dir])</pre> | O(1) |
| <pre> (root → hijos[¬dir]) ← nodo</pre> | O(1) |
| <pre> nodo ← (root → hijos[¬dir])</pre> | O(1) |
| <pre> (root → hijos[¬dir]) ← (nodo → hijos[dir])</pre> | O(1) |
| <pre> (nodo → hijos[dir]) ← root</pre> | O(1) |
| <pre> res ← nodo</pre> | O(1) |
| Complejidad : $O(1)$ | |

| | |
|---|------|
| <pre>ajustarBalance (in/out puntero(nodoAvl): root, in int: dir, in int: bal) → res: puntero(nodoAvl) nodo: puntero(nodoAvl) ← (root → hijos[dir])</pre> | O(1) |
| <pre> nodoHijo: puntero(nodoAvl) ← (nodoUno → hijos[¬dir])</pre> | O(1) |
| <pre> if(nodoHijo → balance = 0) then</pre> | O(1) |
| <pre> (root → balance) ← 0</pre> | O(1) |
| <pre> (nodo → balance) ← 0</pre> | O(1) |
| <pre> else</pre> | |
| <pre> if(nodoHijo → balance = bal) then</pre> | O(1) |
| <pre> (root → balance) ← -bal</pre> | O(1) |
| <pre> (nodo → balance) ← 0</pre> | O(1) |
| <pre> else</pre> | |
| <pre> (root → balance) ← 0</pre> | O(1) |
| <pre> (nodo → balance) ← bal</pre> | O(1) |
| <pre> end if</pre> | |
| <pre> end if</pre> | |
| <pre> (nodoHijo → balance) ← 0</pre> | O(1) |
| Complejidad : $O(1)$ | |

| | |
|--|------|
| <pre>iBorrar (in/out $dicc_{avl}(\alpha)$: tree, in nat: c)</pre> | |
| <pre> if(tree != NULL) then</pre> | O(1) |
| <pre> it: puntero(nodoAvl) ← tree</pre> | O(1) |
| <pre> up: pila(puntero(nodoAvl))</pre> | O(1) |
| <pre> upd: pila(int)</pre> | O(1) |

| | |
|---|-----------|
| break: bool \leftarrow false | O(1) |
| while(break = false) | O(1) |
| if (it \rightarrow clave = c) then | O(1) |
| break \leftarrow true | O(1) |
| end if | |
| if (it \rightarrow clave < c) then | O(1) |
| Apilar(upd, 1) | O(1) |
| else | |
| Apilar(upd, 0) | O(1) |
| end if | |
| Apilar(up, it) | O(1) |
| it \leftarrow (it \rightarrow hijos[Tope(upd)]) | O(1) |
| do | O(log(k)) |
| if((it \rightarrow hijos[0] = NULL) \vee (it \rightarrow hijos[0] = NULL)) then | O(1) |
| if(it \rightarrow hijos[0] = NULL) then | O(1) |
| dir: int \leftarrow 1 | O(1) |
| else | |
| dir: int \leftarrow 0 | O(1) |
| end if | |
| if(Tamano(up) > 1) then | O(1) |
| (Tope(up) \rightarrow hijos[Tope(upd)]) \leftarrow (it \rightarrow hijos[dir]) | O(1) |
| else | |
| tree \leftarrow (it \rightarrow hijos[dir]) | O(1) |
| end if | |
| else | |
| heredero: puntero(nodoAvl) \leftarrow (it \rightarrow hijos[1]) | O(1) |
| Tope(upd) \leftarrow 1 | O(1) |
| Tope(up) \leftarrow it | O(1) |
| while(heredero \rightarrow hijos[0] != null) | O(1) |
| Apilar(upd, 0) | O(1) |
| Apilar(up, heredero) | O(1) |
| heredero \leftarrow (heredero \rightarrow hijos[0]) | O(1) |
| do | O(log(k)) |
| (it \rightarrow clave) \leftarrow (heredero \rightarrow clave) | O(1) |
| Desapilar(up) | O(1) |
| Desapilar(upd) | O(1) |
| if(Tope(up) = it) then | O(1) |
| (Tope(up) \rightarrow hijos[1]) \leftarrow (heredero \rightarrow hijos[1]) | O(1) |
| else | |
| (Tope(up) \rightarrow hijos[0]) \leftarrow (heredero \rightarrow hijos[1]) | O(1) |
| end if | |
| end if | |
| break \leftarrow false | O(1) |
| while((break = false) \wedge (Tamano(up) \geq 0)) | O(1) |
| if(Tope(upd) != 0) then | O(1) |
| (Tope(up) \rightarrow balance) \leftarrow (Tope(up) \rightarrow balance) - 1 | O(1) |
| else | |
| (Tope(up) \rightarrow balance) \leftarrow (Tope(up) \rightarrow balance) + 1 | O(1) |

```

    end if

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

    do
    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])

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

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

    res ← root
    Complejidad :  $O(1)$ 

```

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

```

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

```

```

while (!done) do
  hijoMenor  $\leftarrow$  (actual $\rightarrow$ hijos[0])

  if (hijoMenor  $\neq$  NULL) then
    actual  $\leftarrow$  hijoMenor
  else
    res  $\leftarrow$  (actual $\rightarrow$ data)
    done  $\leftarrow$  true
  end if
end while

```

```

Inorder (in  $dicc_{avl}(\alpha) : n \rightarrow res$ : lista(tupla(clave, significado))
  c: puntero(nodoAvl)  $\leftarrow$  n
  p: pila(puntero(nodoAvl))  $\leftarrow$  Vacía()
  done: bool  $\leftarrow$  false
  res  $\leftarrow$  Vacía()

  while (!done) do
    if (c  $\neq$  NULL) then
      Apilar(p, c)
      c  $\leftarrow$  (c $\rightarrow$ hijos[0])
    else
      if !EsVacía?(p) then
        AgregarAtras(res, << Tope(p) $\rightarrow$ clave, Tope(p) $\rightarrow$ data >>)
        c  $\leftarrow$  Tope(p) $\rightarrow$ hijos[1]
      else
        done  $\leftarrow$  true
      end if
    end if
  end while

```

```

• = • (in  $dicc_{avl}(\alpha) : d1$ , in  $dicc_{avl}(\alpha) : d2 \rightarrow res$ : bool
  res  $\leftarrow$  Inorder(d1) = Inorder(d2)

```

5. Módulo Trie(α)

5.1. Interfaz

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

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

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

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

Complejidad: $O(1)$

Descripción: Crea un diccionario vacío.

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

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

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

Complejidad: $O(L)$

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

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

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 .

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

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

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

Complejidad: $O(L)$

Descripción: Devuelve el significado correspondiente a la clave c .

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

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

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

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

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

Descripción: 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.