

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	3
1.1. Interfaz	3
1.1.1. Operaciones básicas de DCNet	3
1.2. Representación	4
1.2.1. Representación de dcnet	4
1.2.2. Invariante de Representación	6
1.2.3. Función de Abstracción	9
1.3. Algoritmos	9
2. Módulo Red	14
2.1. Interfaz	14
2.2. Representación	15
2.2.1. Estructura	15
2.2.2. Invariante de Representación	15
2.2.3. Función de Abstracción	18
2.3. Algoritmos	18
3. Módulo Cola de mínima prioridad(α)	24
3.1. Especificación	24
3.2. Interfaz	25
3.2.1. Operaciones básicas de Cola de mínima prioridad	25
3.3. Representación	26
3.3.1. Representación de colaMinPrior	26
3.3.2. Invariante de Representación	26
3.3.3. Función de Abstracción	26
3.4. Algoritmos	26
4. Módulo Diccionario logarítmico(α)	28
4.1. Interfaz	28
4.1.1. Operaciones básicas de Diccionario logarítmico(α)	28
4.1.2. Operaciones auxiliares del TAD	29
4.2. Representación	29
4.2.1. Representación de dicLog(α)	29
4.2.2. Invariante de Representación	29
4.2.3. Función de Abstracción	30
4.3. Algoritmos	30
5. Módulo Árbol binario(α)	37
5.1. Interfaz	37
5.1.1. Operaciones básicas de Árbol binario(α)	37
5.2. Representación	38

5.2.1. Representación de $\text{ab}(\alpha)$	38
5.2.2. Invariante de Representación	38
5.2.3. Función de Abstracción	38
5.3. Algoritmos	38
6. Módulo Diccionario $\text{String}(\alpha)$	40
6.1. Interfaz	40

1. Módulo DCNet

1.1. Interfaz

se explica con: DCNET.

géneros: dcnet.

1.1.1. Operaciones básicas de DCNet

INICIARDCNET(**in** r : red) $\rightarrow res$: dcnet

Pre $\equiv \{true\}$

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

Complejidad: $O(n * (n + L))$ donde n es es la cantidad de computadoras y L es la longitud de nombre de computadora mas larga

Descripción: crea una DCNet nueva tomando una red

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

Pre $\equiv \{$

$dcn =_{obs} dcn_0 \wedge$

$\neg((\exists p':paquete)(paqueteEnTransito(dcn, p') \wedge id(p) = id(p') \wedge origen(p) \in computadoras(red(dcn)) \wedge_L$
 $destino(p) \in computadoras(red(dcn)) \wedge_L hayCamino?(red(dcn), origen(p), destino(p))))$

$\}$

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

Complejidad: $O(L + \log(k))$ donde L es la longitud de nombre de computadora mas larga y k es la longitud de la cola de paquetes mas larga

Descripción: crea un nuevo paquete

AVANZARSEGUNDO(**in/out** dcn : dcnet)

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

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

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

Descripción: envia los paquetes con mayor prioridad a la siguiente compu

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

Pre $\equiv \{true\}$

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

Complejidad: $O(1)$

Descripción: devuelve la red de una DCNet

Aliasing: res es una referencia no modificable

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

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

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

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

Descripción: devuelve el camino recorrido por un paquete

Aliasing: res es una referencia no modificable

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

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

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

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

Descripción: devuelve la cantidad de paquetes enviados por una compu

ENESPERA(**in** $dcn: \text{dcnet}$, **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 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 tupla(*topología*: red,
 vectorCompusDCNet: vector(compuDCNet),
 diccCompusDCNet: diccString(puntero(compuDCNet)),
 conjPaquetesDCNet: conj(paqueteDCNet),
 laQueMásEnvio: puntero(compuDCNet))

donde compuDCNet es tupla(*pc*: puntero(compu),
 conjPaquetes: conj(paquete),
 diccPaquetesDCNet: dicc_{avl}(nat, itConj(paqueteDCNet)),
 colaPaquetesDCNet: colaPrioridad(nat, itConj(paqueteDCNet)),
 paqueteAEnviar: itConj(paqueteDCNet), *enviados*: nat)

donde paqueteDCNet es tupla(*it*: itConj(paquete), *recorrido*: lista(compu))

donde paquete es tupla(*id*: nat, *prioridad*: nat, *origen*: compu, *destino*: compu)

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

1.2.2. Invariante de Representación

- (I) Las compus de los elementos de `vectorCompusDCNet` son punteros a todas las compus de la topología
- (II) Las claves de `diccCompusDCNet` son todos los hostnames de la topología
- (III) Los significados de `diccCompusDCNet` son punteros que apuntan a las `compuDCNet` cuyo hostname equivale a su clave en `vectorCompusDCNet`
- (IV) `laQueMásEnvio` 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
 $(\forall c: \text{compu})(c \in \text{computadoras}(e.\text{topologia}) \iff$
 $($
 $(\exists cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \wedge (cd.pc = \text{puntero}(c)) \wedge$
 $(\exists s: \text{string})(\text{def?}(s, e.\text{diccCompusDCNet}) \wedge (s = c.ip)))$
 $)$
 $) \wedge_L$
 $(\forall cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \iff$
 $(\exists s: \text{string})((s = cd.pc \rightarrow ip) \wedge \text{def?}(s, e.\text{diccCompusDCNet}) \wedge_L$
 $\text{obtener}(s, e.\text{diccCompusDCNet}) = \text{puntero}(cd))$
 $) \wedge_L$
 $(\exists cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \wedge_L$
 $* (cd.pc) = \text{compuQueMásEnvió}(e.\text{vectorCompusDCNet}) \wedge e.\text{laQueMásEnvió} = \text{puntero}(cd)) \wedge_L$
 $(\forall cd_1: \text{compuDCNet})(\text{está?}(cd_1, e.\text{vectorCompusDCNet}) \Rightarrow$
 $(\forall p_1: \text{paquete})(p_1 \in cd_1.\text{conjPaquetes} \Rightarrow$
 $(\forall cd_2: \text{compuDCNet})((\text{está?}(cd_2, e.\text{vectorCompusDCNet}) \wedge cd_1 \neq cd_2) \Rightarrow$
 $(\forall p_2: \text{paquete})(p_2 \in cd_2.\text{conjPaquetes} \Rightarrow p_1.id \neq p_2.id)$
 $)$
 $)$
 $) \wedge_L$
 $(\forall cd: \text{compuDCNet})(\text{está?}(cd, e.\text{vectorCompusDCNet}) \Rightarrow$
 $($
 $(\forall p: \text{paquete})(p \in cd.\text{conjPaquetes} \iff$
 $($
 $((p.\text{origen} \in \text{computadoras}(e.\text{topologia}) \wedge p.\text{destino} \in \text{computadoras}(e.\text{topologia}) \wedge$
 $p.\text{destino} \neq *(cd.pc)) \wedge_L$
 $(\exists sc: \text{secu}(\text{compu}))(sc \in \text{caminosMinimos}(e.\text{topologia}, p.\text{origen}, p.\text{destino}) \wedge \text{está?}(*(cd.pc), sc))) \wedge$
 $(\exists n: \text{nat}) ((\text{def?}(n, cd.\text{diccPaquetesDCNet}) \wedge p.id = n) \wedge_L$
 $(\exists pdn: \text{paqueteDCNet})(pdn \in e.\text{conjPaquetesDCNet} \wedge \text{Siguiente}(pdn.it) = p \wedge$
 $((p.\text{origen} = *(cd.pc) \wedge pdn.\text{recorrido} = *(cd.pc) \bullet <>) \vee$
 $(p.\text{origen} \neq *(cd.pc) \wedge pdn.\text{recorrido} \in \text{caminosMinimos}(e.\text{topologia}, p.\text{origen}, *(cd.pc)))) \wedge$
 $\text{Siguiente}(\text{obtener}(n, cd.\text{diccPaquetesDCNet})) = pdn$
 $)$
 $)$
 $) \wedge_L$
 $(\neg \text{vacía?}(cd.\text{colaPaquetesDCNet}) \iff$
 $(\exists p: \text{paquete})((p \in cd.\text{conjPaquetes}) \wedge (p = \text{paqueteMásPrioridad}(cd.\text{conjPaquetes})) \wedge$
 $(\exists pdn: \text{paqueteDCNet})((pdn \in e.\text{conjPaquetesDCNet}) \wedge (\text{Siguiente}(pdn.it) = p) \wedge$
 $(\text{Siguiente}(\text{proximo}(cd.\text{colaPaquetesDCNet})) = pdn))$
 $)$
 $) \wedge_L$
 $(cd.\text{enviados} \geq \text{enviadosCompu}(*(cd.pc), e.\text{vectorCompusDCNet})) \wedge$
 $(\neg \text{HaySiguiente?}(cd.\text{paqueteAEnviar}))$
 $)$

compuQueMásEnvió : 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

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

1.3. Algoritmos

iIniciarDCNet (in topo: red) \rightarrow res: estr

```

res.topologia  $\leftarrow$  Copiar(topo) O(n! * n6)
res.vectorCompusDCNet  $\leftarrow$  Vacía() O(1)
res.diccCompusDCNet  $\leftarrow$  CrearDicc() O(1)
res.laQueMasEnvio  $\leftarrow$  NULL O(1)
res.conjPaquetesDCNet  $\leftarrow$  Vacío() O(1)

it Conj(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)
it Conj(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)

it Conj(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)): intercamios ←
            CrearIt(CaminosMinimos(dcn.topologia,
                *(dcn.vectorCompusDCNet[i].pc), pAEnviar.destino))
        compu: siguientecompu ← Siguiente(itercamios)[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 : 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 ← 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 y recalcula los caminos mínimos de la red.

COMPUTADORAS(in r : red) $\rightarrow res : conj(compu)$
Pre $\equiv \{true\}$
Post $\equiv \{alias(res =_{obs} computadoras(r))\}$
Complejidad: $O(1)$
Descripción: Devuelve el conjunto de computadoras de la red.
Aliasing: El conjunto se da por referencia, y es modificable si y solo si la red es 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

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

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'

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

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

Descripción: Devuelve una copia la red

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

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 `tupla(compus: conj(compu) ,
 dns: diccString(nodoRed))`

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

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

2.2.2. Invariante de Representación

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

- (II) Para cada compu, el diccionario de strings *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 \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 $\{ \neg \emptyset?(\text{secus}) \}$
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,  $\emptyset$ ,  $<>$ ),
                                   longMenorSec(darRutas(nA, nB,  $\emptyset$ ,  $<>$ )))

```

2.2.3. Función de Abstracción

Abs : $\text{estr } e \longrightarrow \text{red}$ {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 ($\exists t1, t2: \text{nat}$) (
 $t1 = \text{interfazUsada}(r, c1, c2) \wedge t2 = \text{interfazUsada}(r, c2, c1) \wedge$
 $\text{def?}(t1, n1.\text{conexiones}) \wedge \text{def?}(t2, n2.\text{conexiones}) \wedge_L$ (
 $\&n2 = \text{obtener}(t1, n1.\text{conexiones}) \wedge \&n1 = \text{obtener}(t2, n2.\text{conexiones})$
)))))

2.3. Algoritmos

```

iIniciarRed ()  $\rightarrow$  res: red
  res.compus  $\leftarrow$  Vacio()

```

O(1)

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

<code>iAgregarComputadora (in/out r: red, in c: compu)</code> <code> itCompus: it Conj(compu) ← CrearIt(r.compus)</code> <code> while HaySiguiente?(itCompus) do</code> <code> nr:nodoRed ← Significado(r.dns, Siguiente(itCompus).ip)</code> <code> Definir(nr.camino, c.ip, Vacio())</code> <code> Avanzar(itCompus)</code> <code> end while</code> <code>AgregarRapido(r.compus, c)</code> <code>Definir(r.dns, compu.ip, Tupla(∞, Vacio(), Vacio(>))</code> <code>InicializarConjCaminos(r, c)</code> Complejidad : $O(n * L)$	$O(1)$ $O(1)$ $O(L)$ $O(L)$ $O(1)$ $O(n * L)$ $O(1)$ $O(L)$ $O(n * L)$
---	--

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

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

<code>CrearTodosLosCaminos (in/out r: red)</code> <code> itCompuA: it Conj(compu) ← CrearIt(r.compus)</code> <code> while HaySiguiente?(itCompuA) do</code> <code> nr:nodoRed ← Significado(r.dns, Siguiente(itCompuA).ip)</code> <code> itCompuB: it Conj(compu) ← CrearIt(r.compus)</code> <code> while HaySiguiente?(itCompuB) do</code> <code> caminimos: conj(lista(compu)) ← Minimos(Caminos</code> <code> (nr, Siguiente(itCompuB).ip)</code> <code> Definir(nr.camino, Siguiente(itCompuB).ip, caminimos)</code> <code> Avanzar(itCompuB)</code> <code> end while</code> <code> Avanzar(itCompuA)</code> <code> end while</code>	$O(1)$ $O(1)$ $O(L)$ $O(1)$ $O(1)$ $O(n! * n * (n + L))$ $O(L)$ $O(1)$ $O(n! * (n^2 * (n + L)))$ $O(1)$ $O(n! * (n^3 * (n + L)))$
--	---

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 ← Vacía()                                       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(1)
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 c0: compu, in c1: 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 c0: compu, in c1: 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 c0: compu, in c1: 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 :  $O(n(L + n(L + n^2)))$ 

```

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

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

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

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

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

$\bullet = \bullet$ (**in** $r_0 : \text{red}$, **in** $r_1 : \text{red}$) $\rightarrow \text{res} : \text{bool}$
 $\text{res} \leftarrow (r_0.\text{compus} = r_1.\text{compus}) \wedge (r_0.\text{dns} = r_1.\text{dns})$
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 $\bullet < : \alpha \times \alpha \longrightarrow \text{bool}$

Relación de orden total estricto¹

géneros $\text{colaMinPrior}(\alpha)$

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

usa **BOOL**

observadores básicos

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

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

$\{\neg \text{vacía?}(c)\}$

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

$\{\neg \text{vacía?}(c)\}$

generadores

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

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

otras operaciones

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

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

PRÓXIMO(in $c : \text{colaMinPrior}(\alpha)$) $\rightarrow res : \alpha$

Pre $\equiv \{\neg \text{vacía?}(c)\}$

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

Complejidad: O(1)

Descripción: Devuelve el próximo elemento a desencolar

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

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

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

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

Complejidad: O(log(tamaño(c)))

Descripción: Quita el elemento más prioritario

Aliasing: Se devuelve el elemento por copia

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

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

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

Complejidad: O(log(tamaño(c)))

Descripción: Agrega al elemento 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(min(tamaño(c), tamaño(c')))

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 `diccLog(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

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

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

3.3.3. Función de Abstracción

$\text{Abs} : \text{estr } e \rightarrow \text{colaMinPrior}$

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

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

3.4. Algoritmos

`iVacía () → res: colaMinPrior(α)`

`res ← CrearDicc()`

$O(1)$

Complejidad : $O(1)$

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

`res ← Vacío?(c)`

$O(1)$

Complejidad : $O(1)$

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

`res ← Proximo(Minimo(c).encolados)`

$O(1)$

Complejidad : $O(1)$

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

res \leftarrow Copiar(Proximo(Minimo(c).encolados))	$O(\text{copy}(\alpha))$
Desencolar(Minimo(c).encolados)	$O(\log(\text{tamaño}(c)))$
if EsVacia?(Minimo(c).encolados) then	$O(1)$
Borrar(c , Minimo(c).prioridad)	$O(\log(\text{tamaño}(c)))$
end if	

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

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

if Definido?(c , p) then	$O(\log(\text{tamaño}(c)))$
Encolar(Obtener(c , p).encolados, a)	$O(\log(\text{tamaño}(c)) + \text{copy}(\alpha))$
else	
nodoEncolados $\text{nuevoNodoEncolados}$	$O(1)$
$\text{nuevoNodoEncolados.encolados} \leftarrow \text{Vacía}()$	$O(1)$
$\text{nuevoNodoEncolados.prioridad} \leftarrow p$	$O(1)$
Encolar($\text{nuevoNodoEncolados.encolados}$, a)	$O(\text{copy}(a))$
Definir(c , p , $\text{nuevoNodoEncolados}$)	$O(\log(\text{tamaño}(c)) + \text{copy}(\text{nodoEncolados}))$
end if	

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

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

res $\leftarrow c_0 = c_1$	$O(\min(\text{tamaño}(c_0), \text{tamaño}(c_1)))$
----------------------------	---

Complejidad : $O(\min(\text{tamaño}(c_0), \text{tamaño}(c_1)))$

4. Módulo Diccionario logarítmico(α)

4.1. Interfaz

se explica con: `DICCIONARIO(NAT, α)`.

géneros: `diccLog(α)`.

4.1.1. Operaciones básicas de Diccionario logarítmico(α)

`CREARDICC()` $\rightarrow res : \text{diccLog}(\alpha)$

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

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

Complejidad: $O(1)$

Descripción: Crea un diccionario vacío

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

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

`DEFINIR(in/out $d : \text{diccLog}(\alpha)$, in $c : \text{nat}$, in $s : \alpha$)`

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

`OBTENER(in/out $d : \text{diccLog}(\alpha)$, in $c : \text{string}$)` $\rightarrow res : \alpha$

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

`BORRAR(in/out $d : \text{diccLog}(\alpha)$, in $c : \text{string}$)`

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

`VACÍO?(in $d : \text{diccLog}(\alpha)$)` $\rightarrow res : \text{bool}$

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

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

Complejidad: $O(1)$

Descripción: Devuelve `true` si y sólo si el diccionario está vacío

`MÍNIMO(in/out $d : \text{diccLog}(\alpha)$)` $\rightarrow res : \alpha$

Pre $\equiv \{\neg \emptyset?(\text{claves}(d))\}$

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

Complejidad: $O(1)$

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

$\bullet = \bullet(\text{in } d: \text{diccLog}(\alpha), \text{in } d': \text{diccLog}(\alpha)) \rightarrow \text{res} : \text{bool}$
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{res} =_{\text{obs}} (d =_{\text{obs}} d')\}$
Complejidad: $O(\max(\#claves(d), \#claves(d')))$
Descripción: Devuelve **true** si y sólo si ambos diccionarios son iguales

4.1.2. Operaciones auxiliares del TAD

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

4.2. Representación

4.2.1. Representación de $\text{diccLog}(\alpha)$

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

donde **estr** es $\text{tupla}(\text{abAvl}: \text{ab}(\text{nodoAvl}), \text{mínimo}: \text{puntero}(\text{ab}(\text{nodoAvl})))$
 donde **nodoAvl** es $\text{tupla}(\text{clave}: \text{nat}, \text{data}: \alpha, \text{balance}: \text{int})$

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 la diferencia en altura entre el hijo derecho y el izquierdo es de uno
 - * -1 si la diferencia en altura entre el hijo izquierdo y el derecho es de uno
- (III) Todas las claves son distintas.
- (IV) El *mínimo* apunta al árbol con el la clave de menor valor, si el diccionario está vacío vale **NULL**.

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

$\text{Rep}(e) \equiv \text{true} \iff \text{esABB}(e.\text{abAvl}) \wedge \text{balanceadoBien}(e.\text{abAvl}) \wedge \text{clavesÚnicas}(e.\text{abAvl}, \text{vacío}) \wedge_{\text{L}}$
 $e.\text{mínimo} = \text{árbolClaveMínima}(e.\text{abAvl})$

$\text{esABB} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{bool}$
 $\text{balanceadoBien} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{bool}$
 $\text{clavesEnÁrbol} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{conj}(\text{nat})$
 $\text{clavesÚnicas} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{bool}$
 $\text{árbolClaveMínima} : \text{ab}(\text{nodoAvl}) \longrightarrow \text{puntero}(\text{ab}(\text{nodoAvl}))$
 $\text{darSignificado} : \text{ab}(\text{nodoAvl}) \ a \times \text{nat} \ c \longrightarrow \alpha \quad \{c \in \text{clavesEnÁrbol}(a) \wedge \text{esABB}(a)\}$

```

esABB(a)           ≡ (¬nil?(a)) ⇒L (
    (¬nil?(izq(a)) ⇒L (raíz(a).clave > raíz(izq(a)).clave ∧ esABB(izq(a)))) ∧
    (¬nil?(der(a)) ⇒L (raíz(a).clave < raíz(der(a)).clave ∧ esABB(der(a))))
)

balanceadoBien(a)  ≡ if (nil?(a)) then
    true
else
    (abs(altura(der(a)) - altura(izq(a))) < 2) ∧
    (raíz(a)→balance = altura(der(a)) - altura(izq(a))) ∧
    balanceadoBien(izq(a)) ∧ balanceadoBien(der(a))
fi

clavesEnÁrbol(a)   ≡ if (nil?(a)) then
    ∅
else
    Ag(raíz(a).clave, (clavesEnÁrbol(izq(a)) ∪ clavesEnÁrbol(der(a))))
fi

clavesÚnicas(a)    ≡ tamaño(a) = #(clavesEnÁrbol(a))

árbolClaveMínima(a) ≡ if (nil?(a)) then
    NULL
else
    if (nil?(izq(a))) then puntero(a) else árbolClaveMínima(izq(a)) fi
fi

darSignificado(a, c) ≡ if (raíz(a).clave = c) then
    raíz(a).data
else
    if (raíz(a).clave < c) then
        darSignificado(izq(a), c)
    else
        darSignificado(der(a), c)
    fi
fi

```

4.2.3. Función de Abstracción

$\text{Abs} : \text{estr } e \longrightarrow \text{dicc}(\text{nat}, \alpha) \quad \{\text{Rep}(e)\}$
 $\text{Abs}(e) =_{\text{obs}} d : \text{dicc}(\text{nat}, \alpha) \mid (\forall n : \text{nat}) ($
 $\quad (\text{def?}(n, d) \Leftrightarrow n \in \text{clavesEnÁrbol}(e.\text{abAvl})) \wedge_L$
 $\quad (\text{def?}(n, d) \Rightarrow_L \text{obtener}(n, d) = \text{darSignificado}(e.\text{abAvl}, n))$
 $\quad)$

4.3. Algoritmos

<pre> iCrearDicc () → res: diccLog(α) res ← <Nil, NULL> Complejidad : O(1) </pre>	O(1)
--	------

<pre> iDefinir (in/out diccLog(α) : d, in nat : c, in α : s) if (Nil?(d.abAvl)) then d.abAvl ← crearArbol(c, s) </pre>	O(1) O(copy(s))
---	--------------------

```

    d.minimo ← puntero(d.abAvl)                                O(1)
else
    it: ab(nodoAvl) ← d.abAvl                                  O(1)
    up: pila(puntero(ab(nodoAvl)))                             O(1)
    upd: pila(bool)                                             O(1)

    Apilar(upd, (Raiz(it).clave < c))                           O(1)
    Apilar(up, puntero(it))                                     O(1)

    while(¬Nil?(subArbol(it, Tope(upd))))                       O(1)
        it ← subArbol(it, Tope(upd))                           O(1)

        Apilar(upd, (Raiz(it).clave < c))                       O(1)
        Apilar(up, puntero(it))                                O(1)
    do                                                         O(log(#claves(d)))

    subArbol(it, Tope(upd)) ← crearArbol(c, s)                  O(copy(s))
    if(c < Raiz(*d.minimo).clave) then                          O(1)
        d.minimo ← puntero(subArbol(it, Tope(upd)))            O(1)
    end if

    break ← false                                              O(1)
    while((Tamano(up) > 0) ∧ ¬break)                             O(1)
        if(Tope(upd)) then                                     O(1)
            Raiz(*Tope(up)).balance ← Raiz(*Tope(up)).balance + 1 O(1)
        else
            Raiz(*Tope(up)).balance ← Raiz(*Tope(up)).balance - 1 O(1)
        end if

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

                if(Tamano(up) > 1) then                          O(1)
                    upTope: puntero(ab(nodoAvl)) ← copy(Tope(up)) O(1)
                    Desapilar(up)                               O(1)
                    Desapilar(upd)                              O(1)
                    subArbol(*Tope(up), Tope(upd)) ← *upTope   O(1)
                else
                    d.abAvl ← *Tope(up)                         O(1)
                end if

                break ← true                                    O(1)
            else
                Desapilar(up)                                   O(1)
                Desapilar(upd)                                  O(1)
            end if
        end if
    do                                                         O(log(#claves(d)))
end if
Complejidad :  $O(\log(\#claves(d))) + O(copy(s))$ 

```

```

crearArbol (in nat: c, in α: s) → res: ab(nodoAvl)
res ← Bin(Nil, <c, copy(s), 0>, Nil)
Complejidad :  $O(copy(s))$ 

```

O(copy(s))


```

subArbol (in/out ab(nodoAvl): a, in bool: dir) → res: ab(nodoAvl)
  if (dir) then
    res ← Der(a)
  else
    res ← Izq(a)
  end if
Complejidad :  $O(1)$ 

```

```

insertarBalance (in/out ab(nodoAvl): root, in bool: dir) → res: ab(nodoAvl)
  hijo: ab(nodoAvl) ← subArbol(root, dir)

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

  if (Raiz(hijo).balance = bal) then
    Raiz(root).balance ← 0
    Raiz(hijo).balance ← 0
    root ← rotacionSimple(root, ¬dir)
  else
    ajustarBalance(root, dir, bal)
    root ← rotacionDoble(root, ¬dir)
  end if

  res ← root
Complejidad :  $O(1)$ 

```

```

rotacionSimple (in/out ab(nodoAvl): root, in bool: dir) → res: ab(nodoAvl)
  hijo: ab(nodoAvl) ← subArbol(root, ¬dir)

  subArbol(root, ¬dir) ← subArbol(hijo, dir)
  subArbol(hijo, dir) ← root

  res ← hijo
Complejidad :  $O(1)$ 

```

```

rotacionDoble (in/out ab(nodoAvl): root, in bool: dir) → res: ab(nodoAvl)
  nieto: ab(nodoAvl) ← subArbol(subArbol(root, ¬dir), dir)

  subArbol(subArbol(root, ¬dir), dir) ← subArbol(nieto, ¬dir)
  subArbol(nieto, ¬dir) ← subArbol(root, ¬dir)
  subArbol(root, ¬dir) ← nieto

  nieto ← subArbol(root, ¬dir)
  subArbol(root, ¬dir) ← subArbol(nieto, dir)
  subArbol(nieto, dir) ← root

  res ← nieto

```

Complejidad : $O(1)$

```

ajustarBalance (in/out ab(nodoAvl): root, in bool: dir, in int: bal)
  hijo: ab(nodoAvl) ← subArbol(root, dir)                                O(1)
  nieto: ab(nodoAvl) ← subArbol(hijo, ¬dir)                             O(1)

  if (Raiz(nieto).balance = 0) then                                     O(1)
    Raiz(root).balance ← 0                                             O(1)
    Raiz(hijo).balance ← 0                                             O(1)
  else
    if (Raiz(nieto).balance = bal) then                                O(1)
      Raiz(root).balance ← -bal                                       O(1)
      Raiz(hijo).balance ← 0                                           O(1)
    else
      Raiz(root).balance ← 0                                           O(1)
      Raiz(hijo).balance ← bal                                         O(1)
    end if
  end if

  Raiz(nieto).balance ← 0                                             O(1)
Complejidad :  $O(1)$ 

```

```

iBorrar (in/out diccLog( $\alpha$ ): d, in nat: c)
  it: ab(nodoAvl) ← d.abAvl                                           O(1)
  padre: ab(nodoAvl) ← Nil                                             O(1)
  padreDir: bool ← false                                              O(1)
  up: pila(puntero(ab(nodoAvl)))                                       O(1)
  upd: pila(bool)                                                      O(1)

  while(Raiz(it).clave ≠ c)                                           O(1)
    Apilar(upd, (Raiz(it).clave < c))                                  O(1)
    Apilar(up, puntero(it))                                           O(1)

    padre ← it                                                         O(1)
    padreDir ← Tope(upd)                                              O(1)

    it ← subArbol(it, Tope(upd))                                       O(1)
  do                                                                    O(log(#claves(d)))

  if (Raiz(it).clave = Raiz(*d.minimo).clave) then                    O(1)
    if (Nil?(padre)) then                                             O(1)
      d.minimo ← NULL                                                O(1)
    else
      d.minimo ← puntero(padre)                                       O(1)
    end if
  end if

  if (Nil?(Izq(it)) ∨ Nil?(Der(it))) then                              O(1)
    dir: bool ← Nil?(Izq(it))                                         O(1)

    if (Tamano(up) > 1) then                                           O(1)
      SubArbol(*Tope(up), Tope(upd)) ← subArbol(it, dir)           O(1)
    else
      d.abAvl ← subArbol(it, dir)                                     O(1)
    end if

```

```

else
    heredero: ab(nodoAvl) ← Der(it)                                O(1)

    Apilar(Tope(upd), true)                                       O(1)
    Apilar(Tope(up), puntero(it))                                O(1)

    while(¬Nil?(Izq(heredero))                                    O(1)
        Apilar(upd, false)                                       O(1)
        Apilar(up, puntero(heredero))                            O(1)
        heredero ← Izq(heredero)                                 O(1)
    do                                                            O(log(#claves(d)))

    subArbol(*Tope(up), Tope(up) = puntero(it)) ← Der(heredero) O(1)

    Izq(heredero) ← Izq(it)                                       O(1)
    Der(heredero) ← Der(it)                                       O(1)

    if(¬Nil?(padre)) then                                         O(1)
        subArbol(padre, padreDir) ← heredero                    O(1)
    end if
end if

break: bool ← false                                             O(1)
while((Tamano(up) > 0) ∧ ¬break)                                  O(1)
    if(Tope(upd)) then                                           O(1)
        Raiz(*Tope(up)).balance ← Raiz(*Tope(up)).balance + 1 O(1)
    else
        Raiz(*Tope(up)).balance ← Raiz(*Tope(up)).balance - 1 O(1)
    end if

    if(abs(Raiz(*Tope(up)).balance) = 1) then                    O(1)
        break ← true                                             O(1)
    else
        if(abs(Raiz(*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(ab(nodoAvl)) ← copy(Tope(up)) O(1)
                Desapilar(up)                                   O(1)
                Desapilar(upd)                                  O(1)
                subArbol(*Tope(up), Tope(upd)) ← *upTope        O(1)
            else
                d.abAvl ← *Tope(up)                               O(1)
            end if
        else
            Desapilar(up)                                       O(1)
            Desapilar(upd)                                       O(1)
        end if
    end if
do                                                            O(log(#claves(d)))
Complejidad :  $O(\log(\#claves(d)) + copy(data))$ 

```

```

removerBalanceo (in/out ab(nodoAvl): root, in bool: dir, in/out puntero(bool): done) → res: ab(nodoAvl)
    hijo: ab(nodoAvl) ← subArbol(root, ¬dir)                    O(1)

    if(dir) then                                                O(1)
        bal ← 1                                                O(1)
    else

```

```

    bal ← -1
end if

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

res ← root
Complejidad :  $O(1)$ 

```

```

iMínimo (in  $diccLog(\alpha)$ : d) → res:  $\alpha$ 
    res ← Raiz(*d.minimo).data
Complejidad :  $O(1)$ 

```

```

iDefinido? (in  $diccLog(\alpha)$ : d, in nat: c) → res: bool
    definido:bool ← false
    it:ab(nodoAvl) ← d.abAvl

    while(¬Nil?(it) ∧ ¬definido) do
        definido ← (Raiz(it).clave = c)
        it ← subArbol(it, (Raiz(it).clave < c))
    end while

    res ← definido
Complejidad :  $O(\log(\#claves(d)))$ 

```

```

iObtener (in/out  $diccLog(\alpha)$ : d, in nat: c) → res:  $\alpha$ 
    it:ab(nodoAvl) ← d.abAvl

    while(Raiz(it).clave ≠ c) do
        it ← subArbol(it, (Raiz(it).clave < c))
    end while

    res ← Raiz(it).data
Complejidad :  $O(\log(\#claves(d)))$ 

```

```

iVacio? (in  $diccLog(\alpha)$ : d) → res: bool
    res ← Nil?(d.abAvl)
Complejidad :  $O(1)$ 

```

```

inorder (in diccLog( $\alpha$ ): d)  $\rightarrow$  res: lista(tupla(nat,  $\alpha$ ))
  root: ab(nodoAvl)  $\leftarrow$  d.abAvl                                O(1)
  p: pila(puntero(ab(nodoAvl)))  $\leftarrow$  Vacía()                    O(1)
  done: bool  $\leftarrow$  false                                         O(1)
  res  $\leftarrow$  Vacía()                                              O(1)

  while (!done) do                                                O(1)
    if ( $\neg$ Nil?(root)) then                                       O(1)
      Apilar(p, puntero(root))                                    O(1)
      root  $\leftarrow$  Izq(root)                                       O(1)
    else
      if  $\neg$ EsVacía?(p) then                                       O(1)
        AgregarAtras(res, <Raiz(*Tope(p)).clave, Raiz(*Tope(p)).data>) O(1)
        root  $\leftarrow$  Der(*Tope(p))                                O(1)
      else
        done  $\leftarrow$  true                                         O(1)
      end if
    end if
  end while
Complejidad :  $O(\#claves(d))$ 

```

```

• = • (in diccLog( $\alpha$ ): d1, in diccLog( $\alpha$ ): d2)  $\rightarrow$  res: bool
  res  $\leftarrow$  inorder(d1) = inorder(d2)                            O(max(#claves(d1), #claves(d2)))
Complejidad :  $O(\max(\#claves(d1), \#claves(d2)))$ 

```

5. Módulo Árbol binario(α)

5.1. Interfaz

se explica con: $\text{ÁRBOL BINARIO}(\alpha)$.

géneros: $\text{ab}(\alpha)$.

5.1.1. Operaciones básicas de Árbol binario(α)

$\text{NIL}() \rightarrow res : \text{ab}(\alpha)$

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

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

Complejidad: $O(1)$

Descripción: Crea un árbol binario nulo

$\text{BIN}(\text{in } i : \text{ab}(\alpha), \text{in } r : \alpha, \text{in } d : \text{ab}(\alpha)) \rightarrow res : \text{ab}(\alpha)$

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

Post $\equiv \{res =_{\text{obs}} \text{bin}(i, r, d)\}$

Complejidad: $O(\text{copy}(r) + \text{copy}(i) + \text{copy}(d))$

Descripción: Crea un árbol binario con hijo izquierdo i , hijo derecho d y raíz de valor r

$\text{RAÍZ}(\text{in/out } a : \text{ab}(\alpha)) \rightarrow res : \alpha$

Pre $\equiv \{\neg \text{nil?}(a)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{raíz}(a))\}$

Complejidad: $O(1)$

Descripción: Devuelve el valor de la raíz del árbol

Aliasing: res es modificable si y sólo si a lo es

$\text{IZQ}(\text{in/out } a : \text{ab}(\alpha)) \rightarrow res : \text{ab}(\alpha)$

Pre $\equiv \{\neg \text{nil?}(a)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{izq}(a))\}$

Complejidad: $O(1)$

Descripción: Devuelve el hijo izquierdo

Aliasing: res es modificable si y sólo si a lo es

$\text{DER}(\text{in/out } a : \text{ab}(\alpha)) \rightarrow res : \text{ab}(\alpha)$

Pre $\equiv \{\neg \text{nil?}(a)\}$

Post $\equiv \{\text{alias}(res =_{\text{obs}} \text{der}(a))\}$

Complejidad: $O(1)$

Descripción: Devuelve el hijo derecho

Aliasing: res es modificable si y sólo si a lo es

$\text{NIL?}(\text{in/out } a : \text{ab}(\alpha)) \rightarrow res : \text{bool}$

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

Post $\equiv \{res =_{\text{obs}} \text{nil?}(a)\}$

Complejidad: $O(1)$

Descripción: Devuelve **true** si res es un árbol vacío

5.2. Representación

5.2.1. Representación de $\text{ab}(\alpha)$

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

donde *estr* es puntero(*nodoAb*)

donde *nodoAb* es tupla(*raiz*: α , *hijos*: arreglo[2] de $\text{ab}(\alpha)$)

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

$\text{Abs} : \text{estr } e \longrightarrow \text{ab}(\alpha) \quad \{\text{Rep}(e)\}$
 $\text{Abs}(e) =_{\text{obs}} \text{abn} : \text{ab}(\alpha) \mid (\text{nil?}(\text{abn}) \Leftrightarrow e = \text{NULL}) \wedge$
 $(\neg \text{nil?}(\text{abn}) \Rightarrow_{\text{L}} (\text{raíz}(\text{abn}) = e \rightarrow \text{raíz} \wedge \text{izq}(\text{abn}) = e \rightarrow \text{hijos}[0] \wedge \text{der}(\text{abn}) = e \rightarrow \text{hijos}[1]))$

5.3. Algoritmos

iNil () \rightarrow res: $\text{ab}(\alpha)$

res \leftarrow NULL

O(1)

Complejidad : $O(1)$

iBin (in $i : \text{ab}(\alpha)$, in $r : \alpha$, in $d : \text{ab}(\alpha)$) \rightarrow res: $\text{ab}(\alpha)$

nuevoAb:nodoAb

O(1)

nuevoAb.raiz \leftarrow copy(r)

O(copy(r))

nuevoAb.hijos[0] \leftarrow copy(i)

O(copy(i))

nuevoAb.hijos[1] \leftarrow copy(d)

O(copy(d))

res \leftarrow puntero(nuevoAb)

O(1)

Complejidad : $O(\text{copy}(r) + \text{copy}(i) + \text{copy}(d))$

iRaíz (in/out $a : \text{ab}(\alpha)$) \rightarrow res: α

res \leftarrow ($a \rightarrow \text{raíz}$)

O(1)

Complejidad : $O(1)$

iIzq (in/out $a : \text{ab}(\alpha)$) \rightarrow res: $\text{ab}(\alpha)$

$\text{res} \leftarrow (a \rightarrow \text{hijos}[0])$

$O(1)$

Complejidad : $O(1)$

iDer (**in/out** $a : \mathbf{ab}(\alpha)$) \rightarrow res: $\mathbf{ab}(\alpha)$

$\text{res} \leftarrow (a \rightarrow \text{hijos}[1])$

$O(1)$

Complejidad : $O(1)$

iNil? (**in** $a : \mathbf{ab}(\alpha)$) \rightarrow res: bool

$\text{res} \leftarrow (a = \text{NULL})$

$O(1)$

Complejidad : $O(1)$

6. Módulo Diccionario $\text{String}(\alpha)$

6.1. Interfaz

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

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

$\text{CREARDICC}() \rightarrow res : \text{diccString}(\alpha)$

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

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

Complejidad: $O(1)$

Descripción: Crea un diccionario vacío.

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

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

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

Complejidad: $O(L)$

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

$\text{DEFINIR}(\text{in } d : \text{diccString}(\alpha), \text{in } c : \text{string}, \text{in } s : \alpha)$

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 .

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

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.

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

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

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

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

Descripción: Indica si d es igual d'

$\text{COPIAR}(\text{in } \text{dicc} : \text{diccString}(\alpha)) \rightarrow res : \text{diccString}(\alpha)$

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

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

Complejidad: $O(n * L * \text{copy}(\alpha))$

Descripción: Devuelve una copia del diccionario