Algoritmos y Estructuras de Datos II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Número 2

DCNet

Grupo: 21

Integrante	LU	Correo electrónico
Alvarez, Lautaro Leonel	268/14	lautarolalvarez@gmail.com
Maddonni, Axel Ezequiel	200/14	axel.maddonni@gmail.com
Thibeault, Gabriel Eric	114/13	grojo94@hotmail.com
Vigali, Leandro Ezequiel	951/12	leandrovigali@yahoo.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo Red

Interfaz

usa: CONJ(α), ITCONJ(α), LISTA(α), ITLISTA(α).

se explica con: RED.

géneros: `red`.

Operaciones de Red

COMPUTADORAS(**in** r : `red`) $\rightarrow$ res : `conj(hostname)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ dameHostnames(computadoras(r))}
Descripción: devuelve el conjunto de las computadoras.
Aliasing: res se devuelve por copia.

CONECTADAS?(**in** r : `red`, **in** $c1$: `hostname`, **in** $c2$: `hostname`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$c1$, $c2 \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ conectadas?(r, dameCompu($c1$), dameCompu($c2$))}
Descripción: indica si las computadoras estan conectadas por alguna de sus interfaces.

INTERFAZUSADA(**in** r : `red`, **in** $c1$: `hostname`, **in** $c2$: `hostname`) $\rightarrow$ res : `interfaz`
Pre $\equiv$ {conectadas?(r, dameCompu($c1$), dameCompu($c2$))}
Post $\equiv$ {$res =_{obs}$ interfazUsada(r, dameCompu($c1$), dameCompu($c2$))}
Descripción: devuelve la interfaz por la cual estan conectadas c1 y c2.

INICIARRED() $\rightarrow$ res : `red`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ iniciarRed()}
Descripción: crear una nueva Red.

AGREGARCOMPU(**in/out** r : `red`, **in** $c1$: `compu`)
Pre $\equiv$ {$r = r_0 \wedge$ ($\forall$ c: compu) $c \in$ computadoras(r_0) $\Rightarrow$ ip(c) $\neq$ $c1$}
Post $\equiv$ {$r =_{obs}$ agregarComputadora(r_0, $c1$)}
Descripción: agregar una computadora a la Red.

CONECTAR(**in** r : `red`, **in** c_1 : `hostname`, **in** i_1 : `interfaz` **in** c_2 : `hostname`, **in** i_2 : `interfaz`)
Pre $\equiv$ {$r = r_0 \wedge c_1, c_2 \in$ dameHostnames(computadoras(r)) $\wedge$ $c_1 \neq c_2 \wedge$ $\neg$conectadas?(r, dameCompu(c_1), dameCompu(c_2)) $\wedge$ $\neg$usaInterfaz?(r, dameCompu(c_1), i_1) $\wedge$ $\neg$usaInterfaz?(r, dameCompu(c_2), i_2) $\wedge$ $i_1 \in$ dameCompu(c_1).interfaces $\wedge$ $i_2 \in$ dameCompu(c_2).interfaces}
Post $\equiv$ {$r =_{obs}$ (conectar(r_0, dameCompu(c_1), i_1, dameCompu(c_2), i_2)}
Descripción: conectar dos computadoras de la red.

VECINOS(**in** r : `red`, **in** c : `hostname`) $\rightarrow$ res : `conj(hostname)`
Pre $\equiv$ {$c \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ dameHostnames(vecinos(r, dameCompu(c)))}
Descripción: da el conjunto de computadoras vecinas.
Aliasing: el conjunto se devuelve por copia.

USAINTERFAZ?(**in** r : `red`, **in** c : `hostname`, **in** i : `interfaz`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$c \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ usaInterfaz?(r, dameCompu(c), i)}
Descripción: indica si la interfaz está siendo utilizada.

CaminosMinimos(**in** r : `red`, **in** c_1 : `hostname`, **in** c_2 : `hostname`) $\rightarrow$ res : `conj(lista(hostname))`
Pre $\equiv$ {$c1,\ c2 \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ dameCaminosdeHostnames(caminosMinimos(r, dameCompu(c_1), dameCompu(c_2)))}
Descripción: devuelve los conjuntos de caminos minimos entre las computadoras ingresadas.
Aliasing: res se devuelve por copia.

HayCamino?(**in** r : `red`, **in** c_1 : `hostname`, **in** c_2 : `hostname`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$c1,\ c2 \in$ dameHostnames(computadoras(r))}
Post $\equiv$ {$res =_{obs}$ hayCamino?(r, dameCompu(c_1), dameCompu(c_2))}
Descripción: indica si las computadoras son alcanzables mediante algún camino.

$\bullet == \bullet$(**in** r_1 : `red`, **in** r_2 : `red`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ ($r_1 =_{obs} r_2$}
Descripción: indica si dos redes son iguales.

donde:
`hostname` es `string`,
`interfaz` es `nat`,
`compu` es `tupla`<ip: `hostname`, interfaces: `conj(interfaz)`>.

Especificación de las operaciones auxiliares utilizadas en la interfaz (no exportadas)

TAD Red extendida

extiende Red

otras operaciones

damehostnames	: conj(compu) $\longrightarrow$ conj(hostname)	
dameCompu	: red r $\times$ hostname s $\longrightarrow$ compu	{$s \in$ hostnames(r)}
auxDameCompu	: red r $\times$ hostname s $\times$ conj(compu) cc $\longrightarrow$ compu	{$s \in$ hostnames(r) $\wedge$ $cc \subset$ computadoras(r)}
dameCaminosDeHostnames	: conj(secu(compu)) $\longrightarrow$ conj(secu(hostname))	
dameSecuDeHostnames	: secu(compu) $\longrightarrow$ secu(hostname)	

axiomas $\forall\ r$: red, $\forall\ cc$: conj(compu), $\forall\ s$: hostname, $\forall\ cs$: conj(secu(compu)), $\forall\ secu$: secu(compu)

dameHostnames(cc) $\equiv$ **if** vacio?(cc) **then**
$\emptyset$
else
Ag(ip(dameUno(cc)), dameHostnames(sinUno(cc)))
fi

dameCompu(r, s) $\equiv$ auxDameCompu(r, s, computadoras(r))

auxDameCompu(r, s, cc) $\equiv$ **if** ip(dameUno(cc)) = s **then**
dameUno(cc)
else
auxDameCompu(r, s, sinUno(cc))
fi

dameCaminosDeHostnames(cs) $\equiv$ **if** vacio?(cs) **then**
$\emptyset$
else
Ag(dameSecuDeHostnames(dameUno(cs)),
dameCaminosDeHostnames(sinUno(cs)))
fi

dameSecuDeHostnames($secu$) $\equiv$ **if** vacia?($secu$) **then**
<>
else
ip(prim($secu$)) • dameSecuDeHostnames(fin($secu$))
fi

Fin TAD

Representación

red se representa con `estr_red`

donde:
`estr_red` es `dicc(hostname, datos)`

donde `datos` es `tupla(`*interfaces*: `conj(interfaz)`
conexiones: `dicc(interfaz, hostname)`
alcanzables: `dicc(dest: hostname, caminos: conj(lista(hostname)))`)

`hostname` es `string`, `interfaz` es `nat`.

Rep : estr_red $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$ ($\forall$ c: hostname, $c \in$ claves(e)) (
\\ *1* claves(obtener(e, c).conexiones) $\subseteq$ obtener(e, c).interfaces $\wedge$
($\forall$ i: interfaz, $i \in$ claves(obtener(e, c).conexiones)) (
\\ *2* obtener(obtener(e, c).conexiones, i) $\in$ claves(e) $\wedge$
\\ *3* obtener(obtener(e, c).conexiones, i) $\neq c$ $\wedge$
\\ *4* ($\neg$ $\exists$ i': interfaz, $i' \in$ claves(obtener(e, c).conexiones), $i \neq i'$) obtener(obtener(e, c).conexiones, i) == obtener(obtener(e, c).conexiones, i') $\wedge$
\\ *5* ($\forall$ h: hostname) (h == obtener(obtener(e, c).conexiones, i) $\Rightarrow$ ($\exists$ i':int) obtener(obtener(e, h).conexiones, i') == c)) $\wedge$
\\ *6* claves(obtener(e, c).alcanzables $\subseteq$ claves(e) $\wedge$
($\forall$ a: hostname, $a \in$ claves(obtener(e, c).alcanzables) (
\\ *7* $a \neq c$ $\wedge$
\\ *8* ($\exists$ s: secu(hostname)) esCaminoVálido(c, a, s) $\wedge$
\\ *9* #obtener(obtener(e, c).alcanzables, a) >0 $\wedge_L$
($\forall$ *camino*: secu(hostname), *camino* $\in$ obtener(obtener(e, c).alcanzables, a) (
\\ *10* esCaminoVálido(c, a, *camino*) $\wedge$
\\ *11* $\neg$($\exists$ *camino*$'$: secu(hostname), *camino* $\neq$ *camino*$'$, esCaminoVálido(c, a, *camino*$'$)) long(*camino*$'$) <long(*camino*) $\wedge$
\\ *12* $\neg$($\exists$ *camino*$'$: secu(hostname), *camino* $\neq$ *camino*$'$, esCaminoVálido(c, a, *camino*$'$), long(*camino* == long(*camino*$'$)) (*camino*$'$ $\notin$ obtener(obtener(e, c).alcanzables, a))
))

La abreviatura *esCaminoValido* usada en el Rep se debe leer: (no son funciones, son abreviaturas para hacer más fácil la lectura)

esCaminoValido ($orig$, $dest$, $secu$) $\equiv$ (prim($secu$) == $orig$ $\wedge$
($\forall$ i: nat, $0 < i <$ long($secu$)) esVecino ($secu[i]$, $secu[i+1]$) $\wedge$
$secu$ [long($secu$)-1] == $dest$ $\wedge$
sinRepetidos($secu$))

Con *esVecino* ($h1$, $h2$) $\equiv$ ($\exists$ i: interfaz) $h2$ == obtener (obtener(e, $h1$).conexiones, i)

\\ Rep en Castellano:
Para cada computadora:
1: Las interfaces usadas pertenecen al conjunto de interfaces de la compu.
2: Los vecinos perteneces a las computadoras de la red.
3: Los vecinos son distintos a la compu actual.
4: Los vecinos no se repiten.
5: Las conexiones son bidireccionales.
6: Los alcanzables pertenecen a las computadoras de la red.
7: Los alcanzables son distintos a la actual.
8: Los alcanzables tienen un camino válido hacia ellos desde la actual.
9: Para cada alcanzable, el conjunto de camiinos válidos no es vacío.
10: Todos los caminos en el diccionario alcanzables son válidos.
11: Los caminos son mínimos.
12: Están todos los mínimos.

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

Abs(e) $\equiv$ r | computadoras (r) = dameComputadoras(e) $\wedge_L$
($\forall$ $c1$, $c2$: compu, $c1$, $c2 \in$ computadoras(r)) conectadas?(r, $c1$, $c2$) = ($\exists$ i: interfaz) ($c2$.ip = obtener(obtener(e, $c1$.ip).conexiones, i)) $\wedge_L$
interfazUsada(r, $c1$, $c2$) = buscarClave (obtener(e, $c1$.ip).conexiones, $c2$.ip)

Especificación de las funciones auxiliares utilizadas en abs

dameComputadoras : dicc(hostname;X) $\longrightarrow$ conj(computadoras)
auxDameComputadoras : dicc(hostname;X) $\times$ conj(hostname) $\longrightarrow$ conj(computadoras)
buscarClave : dicc(interfaz;hostname) $\times$ hostname $\longrightarrow$ interfaz
auxBuscarClave : dicc(interfaz;hostname) $\times$ hostname $\times$ conj(interfaz) $\longrightarrow$ interfaz

axiomas $\forall$ e: dicc(hostname, X), $\forall$ d: dicc(interfaz, hostname), $\forall$ cc: conj(hostname), $\forall$ ci: conj(interfaz), $\forall$ h: hostname

dameComputadoras(e) $\equiv$ auxDameComputadoras(e, claves(e)

auxDameComputadoras(e, cc) $\equiv$ **if** $\emptyset?(cc)$ **then**
$\emptyset$
else
Ag(<dameUno(cc), obtener(e, dameUno(cc)).interfaces>, auxDameComputadoras(e, sinUno(cc)))
fi

buscarClave(d, h) $\equiv$ auxBuscarClave(d, h, claves(d))

auxBuscarClave(d, h, ci) $\equiv$ **if** obtener(d, dameUno(cc)) = h **then**
dameUno(cc)
else
auxBuscarClave(d, h, sinUno(ci))
fi

Algoritmos

Algorithm 1 Implementación de Computadoras

```
function iComputadoras(in r: estr_red)→ res: conj(hostname)
    it ← crearIt(r)                                                          ▷ O(1)
    res ← Vacio()                                  ▷ conjunto                ▷ O(1)
    while HaySiguiente(it) do        ▷ Guarda: O(1)   ▷ El ciclo se ejecuta n veces   ▷ O(n)
        Agregar(res, SiguienteClave(it))                                     ▷ O(1)
        Avanzar(it)                                                          ▷ O(1)
    end while
end function
```

Algorithm 2 Implementación de Conectadas?

```
function iConectadas?(in r: estr_red, in c1: hostname, in c2: hostname)→ res: bool
    it ← CrearIt(Significado(r,c1).conexiones)                               ▷ O(n)
    res ← FALSE                                                              ▷ O(1)
    while HaySiguiente(it) && ¬res do ▷ Guarda: O(1) ▷ El ciclo se ejecuta a lo sumo n-1 veces ▷ O(n)
        if SiguienteClave(it)==c2 then                                       ▷ O(L)
            res ← TRUE
        end if
        Avanzar(it)                                                          ▷ O(1)
    end while
end function
```

Algorithm 3 Implementación de InterfazUsada

```
function iInterfazUsada(in r: estr_red, in c1: hostname, in c2: hostname)→ res: interfaz
    it ← CrearIt(significado(r,c1).conexiones)                               ▷ O(n)
    while HaySiguiente(it) do     ▷ Guarda: O(1)   ▷ El ciclo se ejecuta a lo sumo n-1 veces   ▷ O(n)
        if SiguienteSignificado(it)==c2 then                                 ▷ O(L)
            res ← SiguienteClave(it)                ▷ nat por copia          ▷ O(copiar(nat))
        end if
        Avanzar(it)                                                          ▷ O(1)
    end while
end function
```

Algorithm 4 Implementación de IniciarRed

```
function iIniciarRed()→ res: estr_red
    res ← Vacio()                                  ▷ Diccionario                    ▷ O(1)
end function
```

Algorithm 5 Implementación de AgregarCompu

```
function iAgregarCompu(inout r: estr_red, in c1: compu)
    nuevoDiccVacio ← Vacio()                       ▷ Diccionario                    ▷ O(1)
    DefinirRapido(r, c1.ip, tupla(c1.interfaces, nuevoDiccVacio, nuevoDiccVacio))   ▷ O(Copiar(L))
end function
```

Algorithm 6 Implementación de Conectar

```
function iConectar(inout r: estr_red, in c1: hostname)
end function
```

Algorithm 7 Implementación de Vecinos

```
function iVecinos(inout r: estr_red, in c1: hostname)→ res: conj(hostname)
    it ← CrearIt(Significado(r,c1).conexiones)                                  ▷ O(1) + O(n)
    res ← Vacio()                                  ▷ Conjunto                   ▷ O(1)
    while HaySiguiente(it) do     ▷ Guarda: O(1)   ▷ El ciclo se ejecuta a lo sumo n-1 veces   ▷ O(n)
        AgregarRapido(res, SiguienteSignificado(it))                            ▷ O(L)
        Avanzar(it)                                                             ▷ O(1)
    end while
end function
```

Algorithm 8 Implementación de UsaInterfaz

```
function iUsaInterfaz(in r: estr_red, in c: hostname, in i: interfaz)→ res: bool
    res ← Definido?(Significado(r,c).conexiones,i)                              ▷ O(comparar(nat)*n)
end function
```

Algorithm 9 Implementación de CaminosMinimos

```
function iCaminosMinimos(in r: estr_red, in c1: hostname, in c2: hostname)→ res: conj(lista(hostname))
    itCaminos ← crearIt(Significado(Significado(r,c1).alcanzables, c2))         ▷ O(1) + O(L*n)
    res ← Vacio()                                  ▷ Conjunto                   ▷ O(1)
    while HaySiguiente(itCaminos) do
        AgregarRapido(res, Siguiente(itCaminos))                                ▷ O(1)
        Avanzar(itCaminos)                                                      ▷ O(1)
    end while
end function
```

Algorithm 10 Implementación de HayCamino?

```
function iHayCamino?(in r: estr_red, in c1: hostname, in c2: hostname)→ res: bool
    res ← Definido?(Significado(r,c1).alcanzables, c2)
end function
```

Algorithm 11 Implementación de ==

```
function IIGUALDAD(in r1: estr_red, in r2: estr_red)→ res: bool
    res ← TRUE                                                              ▷ O(1)
    if ¬(#Claves(r1)==#Claves(r2)) then                                     ▷ O(comparar(nat))
        res ← FALSE                                                         ▷ O(1)
    else
        itRed1 ← CrearIt(r1)                                                ▷ O(1)
        while HaySiguiente(itRed1) && res do                                ▷ O(1)
            if ¬(Definido?(r2, SiguienteClave(itRed1)) then                 ▷ O(L*n)
                res ← FALSE                                                 ▷ O(1)
            else
                Compu2 ← Significado(r2,SiguienteClave(itRed1))             ▷ O(L*n)
                Compu1 ← SiguienteSignificado(itRed1)                       ▷ O(1)
                if ¬(Comp1.interfaces == Comp2.interfaces) then             ▷ O(m*m) con m=cantidad de
interfaces
                    res ← FALSE                                             ▷ O(1)
                end if
                if ¬(Comp1.conexiones == Comp2.conexiones) then
                    res ← FALSE                                             ▷ O(1)
                end if
                if ¬(#Claves(Compu1.alcanzables)==#Claves(Compu2.alcanzables)) then
                    res ← FALSE                                             ▷ O(1)
                else
                    itAlc1 ← CrearIt(Compu1.alcanzables)                    ▷ O(1)
                    while HaySiguiente(itAlc1) && res do    ▷ se ejecuta a lo sumo n−1 veces   ▷ O(n)
                        if ¬(Definido?(Comp2.alcanzables, SiguienteClave(itAlc1))) then       ▷ O(m)
                            res ← FALSE
                        else
                            Caminos1 ← SiguienteSignificado(itAlc1)         ▷ O(1)
                            Caminos2 ← Significado(Comp2.alcanzables, itAlc1)   ▷ O(n)
                            if ¬(Longitud(Caminos1) == Longitud(Caminos2)) then   ▷ O(comparar(nat))
                                res ← FALSE
                            else
                                itCaminos1 ← CrearIt(Caminos1)              ▷ O(1)
                                while HaySiguiente(itCaminos1) && res do
                                    itCaminos2 ← CrearIt(Caminos2)          ▷ O(1)
                                    noEncontro ← TRUE                       ▷ O(1)
                                    while HaySiguiente(itCaminos2) && noEncontro do
                                        if Siguiente(itCaminos2) == Siguiente(ItCaminos1) then
                                            noEncontro ← FALSE
                                        end if
                                        Avanzar(itCaminos2)                 ▷ O(1)
                                    end while
                                    if noEncontro then                      ▷ O(1)
                                        res ← FALSE                         ▷ O(1)
                                    end if
                                    Avanzar(itCaminos1)                     ▷ O(1)
                                end while
                            end if
                        end if
                        Avanzar(itAlc1)                                     ▷ O(1)
                    end while
                end if
            end if
            Avanzar(itRed1)                                                 ▷ O(1)
        end while
    end if
end function
```

2. Módulo DCNet

Interfaz

usa: RED, CONJ(α), ITCONJ(α), LISTA(α), ITLISTA(α), DICC$_{TRIE}$(κ, σ), DICC$_{AVL}$(κ, σ), CONJ$_{HEAP}$(α), ITCONJ$_{HEAP}$(α).

se explica con: DCNET.

géneros: `dcnet`.

Operaciones de DCNet

RED(**in** d: `dcnet`) $\rightarrow$ res : `red`
Pre $\equiv$ {true}
Post $\equiv$ {alias($res =_{obs}$ red(d))}
Descripción: devuelve la red asociada.
Aliasing: res no es modificable.

CAMINORECORRIDO(**in** d: `dcnet`, **in** p: `IDpaquete`) $\rightarrow$ res : `lista(hostname)`
Pre $\equiv$ {IDpaqueteEnTransito?(d, p)}
Post $\equiv$ {$res =_{obs}$ dameSecuDeHostnames(caminoRecorrido(d, damePaquete(p)))}
Descripción: devuelve el camino recorrido desde el origen hasta el actual.
Aliasing: res se devuelve por copia.

CANTIDADENVIADOS(**in** d: `dcnet`, **in** c: `hostname`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$c \in$ dameHostnames(computadoras(red(d)))}
Post $\equiv$ {$res =_{obs}$ cantidadEnviados(d, dameCompu(c))}
Descripción: devuelve la cantidad de paquetes enviados por la computadora.

ENESPERA(**in** d: `dcnet`, **in** c: `hostname`) $\rightarrow$ res : `conj(paquete)`
Pre $\equiv$ {$c \in$ dameHostnames(computadoras(red(d)))}
Post $\equiv$ {alias($res =_{obs}$ enEspera(d, dameCompu(c)))}
Descripción: devuelve los paquetes en la cola de la computadora.
Aliasing: res no es modificable.

INICIARDCNET(**in** r: `red`) $\rightarrow$ res : `dcnet`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{obs}$ iniciarDCNet(r)}
Descripción: crea una nueva Dcnet.
Aliasing: la red se agrega por copia.

CREARPAQUETE(**in/out** d: `dcnet`, **in** p: `paquete`)
Pre $\equiv$ {$d = d_0 \wedge \neg($ ($\exists$ p': paquete) (paqueteEnTransito?(d, p') $\wedge$ id(p') = id(p)) $\wedge$
origen(p) $\in$ computadoras(red(d)) $\wedge_L$ destino(p) $\in$ computadoras(red(d)) $\wedge_L$ hayCamino?(red(d), origen(p), destino(p))}
Post $\equiv$ {$d =_{obs}$ crearPaquete(d_0, p)}
Descripción: agrega un paquete a la red.
Aliasing: el paquete se agrega por copia.

AVANZARSEGUNDO(**in/out** d: `dcnet`)
Pre $\equiv$ {$d = d_0$}
Post $\equiv$ {$d =_{obs}$ avanzarSegundo(d_0)}
Descripción: realiza los movimientos de paquetes correspondientes, aplicando los cambios necesarios a la dcnet.

PAQUETEENTRANSITO?(**in** d: `dcnet`, **in** p: `IDpaquete`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}

Post $\equiv \{res =_{obs} \text{IDpaqueteEnTransito?}(d, p)\}$
Descripción: indica si el paquete esta en alguna de las colas dado el ID.

LaQueMasEnvio(**in** d: `dcnet`) $\rightarrow$ res : `hostname`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{obs} \text{laQueMasEnvio}(d).\text{ip}\}$
Descripción: devuelve la computadora que más paquetes envió.
Aliasing: res se devuelve por copia.

$\bullet = \bullet$(**in** d_1 : `dcnet`, **in** d_2 : `dcnet`) $\rightarrow$ res : `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{obs} (d_1 =_{obs} d_2\}$
Descripción: indica si dos dcnet son iguales.

donde:
`hostname` es `string`,
`interfaz` es `nat`,
`IDpaquete` es `nat`,
`compu` es `tupla`<ip: `hostname`, interfaces: `conj(interfaz)`>,
`paquete` es `tupla`<id: `IDpaquete`, prioridad: `nat`, origen: `hostname`, destino: `hostname` >.

Especificación de las operaciones auxiliares utilizadas en la interfaz

TAD DCNet extendida

extiende DCNet

otras operaciones

(no exportadas)

damehostnames	: conj(compu)	$\longrightarrow$ conj(hostname)
dameCompu	: dcnet d $\times$ hostname s	$\longrightarrow$ compu $\{s \in \text{dameHostnames}(\text{computadoras}(\text{red}(d)))\}$
auxDameCompu	: hostname s $\times$ conj(compu) cc	$\longrightarrow$ compu
dameSecuDeHostnames	: secu(compu)	$\longrightarrow$ secu(hostname)
IDpaqueteEnTransito?	: dcnet d $\times$ IDpaquete p	$\longrightarrow$ bool
damePaquete	: dcnet d $\times$ IDpaquete p	$\longrightarrow$ paquete $\{\text{IDpaqueteEnTransito?}(d,p)\}$
dameIDpaquetes	: conj(paquete)	$\longrightarrow$ conj(IDpaquete)

axiomas $\forall$ d: dcnet, $\forall$ s: hostname, $\forall$ p: IDpaquete, $\forall$ cc: conj(compu), , $\forall$ $secu$: secu(compu), $\forall$ cp: conj(paquete),

$\text{dameHostnames}(cc) \equiv$ **if** vacio?(cc) **then** $\emptyset$ **else** Ag(ip(dameUno(cc)), dameHostnames(sinUno(cc))) **fi**

$\text{dameCompu}(d, s) \equiv \text{auxDameCompu}(s, \text{computadoras}(\text{red}((d))))$

$\text{auxDameCompu}(s, cc) \equiv$ **if** ip(dameUno(cc)) $= s$ **then** dameUno(cc) **else** auxDameCompu(s, sinUno(cc)) **fi**

dameSecuDeHostnames($secu$) $\equiv$ **if** vacia?($secu$) **then**
<>
else
ip(prim($secu$)) • dameSecuDeHostnames(fin($secu$))
fi

IDpaqueteEnTransito?(d, p) $\equiv$ auxIDpaqueteEnTransito(d, computadoras(red(d)), p)

auxIDpaqueteEnTransito(d, cc, p) $\equiv$ **if** vacio?(cc) **then**
false
else
if $p \in$ dameIDpaquetes(enEspera(dameUno(cc))) **then**
true
else
auxIDpaqueteEnTransito(d, sinUno(cc), p)
fi
fi

dameIDpaquetes(cp) $\equiv$ **if** vacio?(cp) **then**
$\emptyset$
else
Ag(id(dameUno(cp)), dameIDpaquetes(sinUno(cp)))
fi

Fin TAD

Representación

`dcnet` **se representa con** `estr_dcnet`

donde `estr_dcnet` es `tupla`(*red*: `red`
computadoras: `dicc(hostname, X)`
porHostname: $\texttt{dicc}_{TRIE}$ `(hostname, itDicc(hostname, X))`
conMasEnvios: `itDicc(hostname, X)`
caminos: `arreglo_dimensionable de arreglo_dimensionable de lista(hostname)`)

donde `X` es `tupla`(*indice*: `nat`
paquetes: `conj(paquete)`
cola: $\texttt{conj}_{HEAP}$`(itConj(paquete))`
paqPorID: $\texttt{dicc}_{AVL}$ `(IDpaquete, itConj(paquete))`
cantEnvios: `nat`)

Rep : estr_dcnet $\longrightarrow$ bool

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

Abs : estr_dcnet d $\longrightarrow$ dcnet {Rep(d)}

Abs(d) $\equiv$ ...

Algoritmos

Algorithm 12 Implementación de Red

function IRED(**in** d: `estr_dcnet`)→ res: Red
    res ← d.red
end function

Algorithm 13 Implementación de CaminoRecorrido

function ICAMINORECORRIDO(**in** d: `estr_dcnet`, **in** p: `IDPaquete`)→ res: lista(hostname)
    itCompu ← CrearIt(d.computadoras) ▷ O(1)
    yaEncontrado ← FALSE ▷ O(1)
    while HaySiguiente(itCompu) && ¬yaEncontrado **do** ▷ Guarda: O(1) ▷ Se repite a lo sumo n veces ▷ O(n)
        if Definido?(SiguienteSignificado(itCompu).paqPorID, p) **then** ▷ O(log(k))
            paquete ← Significado(SiguienteSignificado(itCompu).paqPorID, p) ▷ O(1)
            yaEncontrado ← TRUE ▷ O(1)
        else
            Avanzar(itCompu) ▷ O(1)
        end if
    end while
    res ← caminos[Significado(d.computadoras, $\pi 3$(paquete)).indice][SiguienteSignificado(itCompu).indice] ▷ O(1) + O(n) + O(1)
end function

Algorithm 14 Implementación de paquetes enviados

function CANTIDADENVIADOS(**in** d: `estr_dcnet`, **in** c: `hostname`)→ res: nat
    $it \leftarrow Significado(d.porHostname, c)$ ▷ O(L)
    $res \leftarrow SiguienteSignificado(it).cantEnvios$ ▷ O(1)
end function

Algorithm 15

function IENESPERA(**in** d: `estr_dcnet`, **in** c: `hostname`)→ res : $estr$
  $it \leftarrow Significado(d.porHostname, c)$ ▷ O(L)
  $res \leftarrow SiguienteSignificado(it).paquetes$ ▷ O(1)
end function

Algorithm 16 Implementación de iniciarDCNet

function INICIARDCNET(**in** r : `red`)→ res: $estr\_dcnet$
  $res.red \leftarrow r$ ▷ Hay que copiar Red?O(copy(r)
  $diccCompus \leftarrow Vaco()$ ▷ O(1)
  $diccHostname \leftarrow Vaco()$ ▷ O(1)
  $index \leftarrow 0$ ▷ O(1)
  $itHostname \leftarrow CrearIt(Computadoras(r))$ ▷ O(1)
  $res.conMasEnvos \leftarrow Siguiente(itHostname)$ ▷ O(1)
  while HaySiguiente(itHostname) **do** ▷ O(#Computadoras(r))
    $X \leftarrow < index, Vaco(), Vaco(), Vaco(), 0 >$ ▷ O(n) segun el apunte
    $itX \leftarrow Definir(diccCompus, Siguiente(itHostname), X)$ ▷ O(copy(Siguiente(itHostname) + copy(X))
    $Definir(diccHostname, Siguiente(itHostname), itX)$ ▷ O(copy(Siguiente(itHostname) + copy(X))
    $index \leftarrow index + 1$ ▷ O(1)
    $Avanzar(itHostname)$ ▷ O(1)
  end while
  $res.computadoras \leftarrow diccCompus$ ▷ O(1)
  $res.porHostaname \leftarrow diccHostname$ ▷ O(1)
  $n \leftarrow \#Claves(diccCompus)$ ▷ O(1)
  $arrayCaminos \leftarrow CrearArreglo(n)$ ▷ O(n)
  $itPC \leftarrow CrearIt(diccCompus)$ ▷ O(1)
  $itPC2 \leftarrow CrearIt(diccCompus)$ ▷ O(1)
  while HaySiguiente(itPC) **do** ▷ O(1)
    $arrayDestinos \leftarrow CrearArreglo(n)$ ▷ O(n)
    while HaySiguiente(itPC2) **do** ▷ O(1)
      $itConj \leftarrow CaminosMinimos(r, SiguienteClave(itPC), SiguienteClave(itPC2))$ ▷ O(???)
      if HaySiguiente(itConj) **then** ▷ O(1)
        $arrayDestinos[SiguienteSignificado(itPC2).indice] \leftarrow Siguiente(itConj)$ ▷ el apunte dice que no es modificable O(1)
      else
        $arrayDestinos[SiguienteSignificado(itPC2).indice] \leftarrow Vaco$ ▷ es necesario esto?O(1)
      end if
      $Avanzar(itPC2)$ ▷ O(1)
    end while
    $arrayCaminos[SiguienteSignificado(itPC).indice] \leftarrow arrayDestinos$ ▷ O(1)
    $Avanzar(itPC)$ ▷ O(1)
  end while
  $res.caminos \leftarrow arrayCaminos$ ▷ O(1)
end function

Algorithm 17

function CREARPAQUETE(**in/out** d: `estr_dcnet`, **in** p: `paquete`)
  $itPC \leftarrow Significado(d.porHostname, paquete.origen)$ ▷ O(L)
  $itPaq \leftarrow Agregar(SiguienteSignificado(itPC).paquetes, p)$ ▷ O(copy(p))
  $Encolar(SiguienteSignificado(itPC).cola, p.prioridad, itPaq)$ ▷ O(log(n)), n cantidad de nodos)
  $Definir(SiguienteSignificado(itPC).paquetesPorID, IDpaquete, itPaq)$ ▷ O(log(n)), n cantidad de nodos
end function

Algorithm 18 Implementación de PaqueteEnTransito?

```
function iPaqueteEnTransito?(in d: estr_dcnet, in p:IDpaquete)→ res: bool
    res ← false                                                          ▷ O(1)
    itCompu ← crearIt(d.computadoras)                                    ▷ O(1)
    while HaySiguiente(itCompu) && ¬res do
▷ a lo sumo n veces, la guarda es O(1)
        itPaq ← crearIt(siguienteSignificado(itCompu).paquetes)          ▷ O(1)
        while (HaySiguiente(itPaq) && Siguiente(itPaq).id ≠ p) do        ▷ a lo sumo k veces, la guarda es
O(1)
            Avanzar(itPaq)                                               ▷ O(1)
        end while
        if Siguiente(itPaq) == p) then                                   ▷ O(1)
            res ← True                                                   ▷ O(1)
        end if
        Avanzar(itCompu)                                                 ▷ O(1)
    end while
end function
```

Algorithm 19 Implementación de LaQueMasEnvió

```
function iLaQueMasEnvió(in d: estr_dcnet)→ res: hostname
    res ← SiguienteClave(d.conMasEnvios)
end function
```

Algorithm 20 Implementación de AvanzarSegundo

```
function IAvanzarSegundo(inout d: estr_dcnet)
    arreglo ← crearArreglo[#Claves(d.computadoras)] de tupla(usado: bool, paquete: paquete, destino: string), donde paquete es tupla(IDpaquete: nat, prioridad: nat, origen: string, destino: string)
                                                    ▷ O(n) para calcular cantidad de claves, O(1) para crearlo
    for (int i=0, < #Claves(d.computadoras), i++) do                    ▷ el ciclo se hará n veces
        arreglo[i].usado = false                                        ▷ O(1)
    end for

\\ Inicializo Iterador
    itCompu ← crearIt(d.computadoras)                                   ▷ O(1)
    i ← 0
                                                    ▷ Ciclo 1: Desencolo y guardo en arreglo auxiliar.
    while (HaySiguiente(itCompu)) do                                    ▷ el ciclo se hará a lo sumo n veces
        if (¬(Vacia?(SiguienteSignificado(itCompu).cola))) then         ▷ O(1)
\\ Borro el de mayor priorirdad del heap:
            itPaquete ← Desencolar(SiguienteSignificado(itCompu).cola)  ▷ O(log k)
\\ Lo elimino del dicc AVL
            Borrar(SiguienteSignificado(itCompu).paquetesPorID, Siguiente(itPaquete).IDpaquete)
                                                                        ▷ O(log k)
\\ Guardo el paquete en una variable
            paqueteDesencolado ← Siguiente(itPaquete)                   ▷ O(1)
\\ Lo elimino del conjunto lineal de paquetes
            EliminarSiguiente(itPaquete)                                ▷ O(1)
\\ Calculo proximo destino fijandome en la matriz
\\ El origen lo tengo en O(1) en el significado del iterador de compus.
            origen ← (SiguienteSignificado(itCompu)).indice             ▷ O(1)
\\ El destino lo obtengo en O(L) buscando por hostname el destino del paquete, y luego guardo el indice.
            itdestino ← Significado(d.porHostname, paqueteDesencolado.destino)  ▷ O(L)
            destino ← (SiguienteSignificado(itdestino)).indice          ▷ O(1)
            proxDest ← d.caminos[origen][destino][1]                    ▷ O(1)
\\ Lo inserto en el arreglo junto con el destino sólo si el destino no era el final.
            if (proxDest ≠ paqueteDesencolado.destino) then
                arreglo[i] ← <true, paqueteDesencolado, proxDest>       ▷ O(1)
            end if
\\ Aumento cantidad de envíos
            SiguienteSignificado(itCompu).cantEnvios ++                 ▷ O(1)
\\ Actualizo conMasEnvios
            envios ← SiguienteSignificado(itCompu).cantEnvios           ▷ O(1)
            if (envios > SiguienteSIgnificado(d.conMasEnvios).cantEnvios) then  ▷ O(1)
                d.conMasEnvios ← itCompu
            end if
        end if
\\ Avanzo de computadora
        Avanzar(itCompu)                                                ▷ O(1)
        i++
    end while
```

```
                                                ▷ Ciclo 2: Encolo los paquetes del vector a sus destinos correspondientes.
    i ← 0
    while HaySiguiente(itCompu) do                          ▷ el ciclo se hará a lo sumo n veces
        if arreglo[i].usado then
\\ Busco el proxDestino guardado en el arreglo por hostname.
            itdestino ← Significado(d.porHostname, arreglo[i].destino)                    ▷ O(L)
\\ Agrego el paquete al conjunto de paquetes del prox destino.
            itpaquete ← AgregarRapido(SiguienteSignificado(itdestino).paquetes, arreglo[i].paquete)
                                                                                          ▷ O(1)
\\ Encolo el heap del destino
            prioridad ← (arreglo[i].paquete).prioridad
            Encolar(SiguienteSignificado(itdestino).cola, prioridad, itpaquete)           ▷ O(log k)
\\ Lo agrego en el dicc AVL.
            IDpaq ← (arreglo[i].paquete).IDpaquete                                        ▷ O(1)
            Definir(SiguienteSignificado(itdestino).paquetesPorID, IDpaq, itpaquete)      ▷ O(log k)
        end if
        i++
        Avanzar(itCompu)
    end while
end function
```

Algorithm 21 Implementación de ==

```
function IIGUALDAD(in d1: estr_dcnet, in d2: estr_dcnet)→ res: bool
\\ Comparo redes usando == de red
    res ← (d1.red == d2.red)                                                              ▷ O(???)
    if (res) then                                                                         ▷ O(1)
        itCompu ← crearIt(d1.computadoras)                                                ▷ O(1)
        string host                                                                       ▷ O(1)
\\ Recorro las computadoras
        while (HaySiguiente(itCompu) && res) do                 ▷ itero O(n) veces, la guarda es O(1)
            host ← SiguienteClave(itCompu)                                                ▷ O(1)
\\ Comparo enEspera usando == de conjunto lineal, y cant. enviados
            res ← (enEspera(d1, host) == enEspera(d2, host) &&
cantidadEnviados(d1,host) == cantidadEnviados(d2,host))                                   ▷ O(???)
            itpaq ← crearIt(SiguienteSignificado(itCompu).paquetes)                       ▷ O(1)
            int j ← 0                                                                     ▷ O(1)
            nat id                                                                        ▷ O(1)
\\ Recorro paquetes de cada computadora
            while (HaySiguiente(itpaq) && res ) do              ▷ itero O(k) veces, la guarda es O(1)
                id ← Siguiente(itpaq).IDpaquete                                           ▷ O(1)
\\ Comparo caminosRecorridos usando == de listas enlazadas
                res ← (caminoRecorrido(d1, id) == caminoRecorrido(d2, id))                ▷ O(???)
                avanzar(itpaq)                                                            ▷ O(1)
            end while
            avanzar (itCompu)                                                             ▷ O(1)
        end while
    end if
end function
```

3. Módulo Cola de Prioridad $_{HEAP}$ (α)

Interfaz

usa: TUPLA, NAT, BOOL, α.

se explica con: COLA DE PRIORIDAD ALTERNATIVA.

géneros: `heap`.

Operaciones de Cola de Prioridad $_{HEAP}$

VACIA() $\rightarrow$ *res* : `heap`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ vacia}
Complejidad: O(1)
Descripción: Crea una cola vacia.

VACIA?(**in** *estr*: `heap`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{obs}$ vacia?(*estr*)}
Complejidad: O(1)
Descripción: Indica si la cola esta vacia.

ENCOLAR(**in/out** *estr*: `heap`, **in** *prio*: `nat`, **in** *valor*: α) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {estr = estr$_0$}
Post $\equiv$ {*res* $\wedge$ estr $=_{obs}$ encolar(estr$_0$)}
Complejidad: O(log(n))
Descripción: Crea un nuevo elemento con los parametros dados y lo agrega a la cola.

DESENCOLAR(**in/out** *estr*: `heap`) $\rightarrow$ *res* : α
Pre $\equiv$ {estr = $estr_0$ $\wedge \neg Vacia?(estr)$}
Post $\equiv$ {estr $=_{obs}$ desencolar($estr_0$) $\wedge$ *res* $=_{obs}$ $proximo(estr_0)$}
Complejidad: O(log(n))
Descripción: Devuelve al elemento de mayor prioridad y lo remueve de la cola. La cola no debe estar vacía.

TAD COLA DE PRIORIDAD ALTERNATIVA(α)

géneros colaPrio(α)

exporta colaPrio(α), generadores, observadores

igualdad observacional
$$\begin{pmatrix}(\forall c_1, c_2 : colaPrio(\alpha))(c_1 =_{obs} c_2 \iff (\forall n : nat)\ conjPorPrio(c_1, n) =_{obs} \\ conjPorPrio(c_2, n))\end{pmatrix}$$

usa BOOL, NAT, CONJUNTO(α)

observadores básicos

prioridades	: colaPrio(α)	$\longrightarrow$	conj(nat)	
conjPorPrio	: colaPrio(α) $\times$ nat	$\longrightarrow$	conj(α)	
próximo	: colaPrio(α) c	$\longrightarrow$	α	{$\neg\emptyset$?(prioridades(c))}

generadores

vacía	:	$\longrightarrow$	colaPrio(α)

encolar : colaPrio(α) $\times$ nat $\times$ α $\longrightarrow$ colaPrio(α)

desencolar : colaPrio(α) c $\longrightarrow$ colaPrio(α) $\{\neg\emptyset?(\text{prioridades}(c))\}$

otras operaciones

próximaPrio : colaPrio(α) $\longrightarrow$ nat

maxConjunto: conjunto(nat) C $\longrightarrow$ nat

axiomas $\forall\, c :\ colaPrio(\alpha), \forall\, n,\ p :\ nat, \forall\, val :\ \alpha, \forall\, conj :\ conj(nat)$

próximaPrio(c) $\equiv$ maxConjunto($prioridades(c)$)

maxConjunto($conj$) $\equiv$ **if** $\emptyset?$(conj) **then** 0 **else if** dameUno($conj$) $\geq$ maxConjunto($sinUno(conj)$) **then** dameUno($conj$) **else** maxConjunto($sinUno(conj)$)

prioridades($vacia$) $\equiv$ $\emptyset$

prioridades($encolar(c, n, val)$) $\equiv$ Ag($n, prioridades(c)$)

prioridades($desencolar(c)$) $\equiv$ **if** $\#conjPorPrio(c, proximaPrio(c)) = 1$ **the** prioridades(c) - {próximaPrio(c)} **else** prioridades(c)

conjPorPrio($vacia, p$) $\equiv$ $\emptyset$

conjPorPrio($encolar(c, n, val), p$) $\equiv$ **if** $p = n$ **then** Ag(val, conjPorPrio(c, p) **else** conjPorPrio(c, p)

conjPorPrio($desencolar(c), p$) $\equiv$ **if** $p = proximaPrio(c)$ **then** sinUno($conjPorPrio(c, p)$) **else** conjPorPrio(c, p)

próximo(encolar(c, n, val)) $\equiv$ **if** $n = proximaPrio(c)$ **then** dameUno($Ag(conjPorPrio(c, n))$) **else** dameUno($conjPorPrio(c, proximaPrio(c))$)

próximo(desencolar(c)) $\equiv$ **if** $\#conjPorPrio(c, proximaPrio(c)) = 1$ **then** dameUno($conjPorPrio(c, maxConjunto(prioridades(c) - \{proximaPrio(c)\}))$) **else** dameUno($sinUno(conjPorPrio(c, proximaPrio(c)))$)

Fin TAD

Representación

heap se representa con estr

donde estr es tupla(*size*: nat
primero: nodo(α)
)

donde nodo(α) es tupla(*padre*: puntero(nodo(α))
izq: puntero(nodo(α))
der: puntero(nodo(α))
prio: nat
valro: α
)

Rep: estr $\rightarrow$ bool
Rep(e) $\equiv$ true $\iff$ $size = \#arbol(estr.primero) \ \wedge_L$
$((estr.primero).padre = null \wedge$
$(\forall e : estr)(e \in arbol(estr.primero) \wedge \ e \neq (estr.primero) \Rightarrow (e.padre \neq null \wedge_L (((e.padre).izq = e \vee (e.padre).der = e) \wedge \neg(((e.padre).izq = e \wedge (e.padre).der = e))))) \wedge$
$(\forall e : estr)(e \in arbol(estr.primero) \Rightarrow ((estr.izq \neq null \Rightarrow estr.prio \geq (estr.izq).prio) \wedge (estr.der \neq null \Rightarrow estr.prio \geq (estr.der).prio))) \wedge$
$(\forall e : estr)(e \in arbol(estr.primero) \Rightarrow caminoHastaRaiz(e, arbol(estr.primero)) \leq \lfloor log_2(size) \rfloor + 1))$

Abs: estr d $\rightarrow$ colaPrio(α) { Rep(d) }
Abs($Vacia()$) $\equiv$ Vacía
$(\forall e : \ estr, n : \ nat, valor : \ \alpha)$(Abs($encolar(e, n, valor)$) $\equiv$ encolar($Abs(e), n, valor$)
$(\forall e : \ estr)$(Abs($desencolar(e)$) $\equiv$ desencolar($Abs(e)$)

Especificación de las operaciones auxiliares utilizadas para Rep y Abs

arbol : nodo(α) $\longrightarrow$ conj(nodo(α))

caminoHastaRaiz : nodo(α) $\longrightarrow$ nat

arbol(n) $\equiv$ **if** $n.izq \neq null \wedge n.der \neq null$ **then** Ag($n.valor, arbol(n.izq) \cup arbol(n.der)$) **else if** $n.izq \neq null$ **then** Ag($n.valor, arbol(n.izq)$) **else if** $n.der \neq null$ **then** Ag($n.valor, arbol(n.der)$) **else** Ag($n.valor, \emptyset$)

caminoHastaRaiz(n) $\equiv$ **if** $n.padre = null$ **then** 0 **else** $caminoHastaRaiz(n.padre) + 1$

Algoritmos

Algorithm 22 Implementación de Vacia

function IVACIA$\rightarrow$ *res*: heap
$res \leftarrow <0, null>$ ▷ O(1)
end function

Algorithm 23 Implementación de Vacia?

function IVACIA?(**in** *estr* : heap)$\rightarrow$ *res*: bool
$res \leftarrow$ $(estr.primero == null)$ ▷ O(1)
end function

Algorithm 24 Implementación de Encolar

```
function iEncolar(in/out estr: heap, in prio: nat, in valor: α)→ res: bool
    res ← true                                                              ▷ O(1)
    if estr.size == 0 then                                                  ▷ O(1)
        estr.primero ← <null, null, null, prio, valor>                      ▷ O(1)
    else
        size + +                                                            ▷ O(1)
        x ← valor                                                           ▷ O(1)
        y ← <>                                                              ▷ O(1)
        while x ≠ 0 do ▷ La cantidad de veces que se ejecuta el ciclo es igual a la altura del heap. Al ser un arbol binario completo, la altura siempre será O(log(n))
            y ← (x%2) • y                                                   ▷ O(1)
            x ← x/2                                                         ▷ O(1)
        end while
        y ← com(y)                                                          ▷ O(log(n))
        z ← estr.primero                                                    ▷ O(1)
        y ← fin(y)                                                          ▷ O(1)
        while long(y)> 1 do                                                 ▷ El ciclo se ejecuta O(log(n)) veces
            z ← if prim(y) == 0 then z.izq else z.der                       ▷ O(1)
            y ← fin(y)                                                      ▷ O(1)
        end while
        w ← <null, null, null, prio, valor>                                 ▷ O(1)
        w.padre ← z                                                         ▷ O(1)
        if prim(y) == 0 then                                                ▷ O(1)
            z.izq ← w                                                       ▷ O(1)
        else
            z.der ← w                                                       ▷ O(1)
        end if
        while w ≠ estr.primero ∧_L w.prio > (w.padre).prio do ▷ La cantidad de veces que se ejecuta el ciclo es a lo sumo la altura del heap, que es O(log(n))
            aux ← w.valor                                                   ▷ O(1)
            w.valor ← (w.padre).valor                                       ▷ O(1)
            (w.padre).valor ← aux                                           ▷ O(1)
            w ← w.padre                                                     ▷ O(1)
        end while
    end if
end function
```

Algorithm 25 Implementación de Desencolar

function IDESENCOLAR(**in/out** *estr*: heap)→ *res*: α
  $res \leftarrow estr.primero$ ▷ O(1)
  $x \leftarrow valor$ ▷ O(1)
  $y \leftarrow <>$ ▷ O(1)
  while x $\neq 0$ **do** ▷ La cantidad de veces que se ejecuta el ciclo es igual a la altura del heap. Al ser un arbol binario completo, la altura siempre será O(log(n))
    $y \leftarrow (x\,\%2) \bullet y$ ▷ O(1)
    $x \leftarrow x/2$ ▷ O(1)
  end while
  $y \leftarrow \text{com}(y)$ ▷ O(log(n))
  $z \leftarrow estr.primero$ ▷ O(1)
  $y \leftarrow \text{fin}(y)$ ▷ O(1)
  while long(y)> 1 **do** ▷ El ciclo se ejecuta O(log(n)) veces
    $z \leftarrow$ **if** prim(y) == 0 **then** $z.izq$ **else** $z.der$ ▷ O(1)
    $y \leftarrow \text{fin}(y)$ ▷ O(1)
  end while
  $w \leftarrow <null, null, null, prio, valor>$ ▷ O(1)
  $w.padre \leftarrow z$ ▷ O(1)
  if prim(y) == 0 **then** ▷ O(1)
    $z.izq \leftarrow w$ ▷ O(1)
  else
    $z.der \leftarrow w$ ▷ O(1)
  end if
  $(estr.primero).valor \leftarrow z.valor$ ▷ O(1)
  borrar(z) ▷ O(1)
  $size--$ ▷ O(1)
  while $(z.izq \neq null \ \vee z.der \neq null) \ \wedge_L \ z.valor < maxValor(z.izq, z.der)$ **do** ▷ La cantidad de veces que se ejecuta el ciclo es a lo sumo la altura del heap, que es O(log(n))
    ▷ maxValor devuelve el maximo valor si ambos punteros son validos, o el valor apuntado por el puntero no nulo en caso de que alguno no lo sea
  end while
  if $z.der == null \ \vee_L \ (z.izq).valor \geq (z.der).valor$ **then** ▷ O(1)
    $aux \leftarrow z.valor$ ▷ O(1)
    $z.valor \leftarrow (z.izq).valor$ ▷ O(1)
    $(z.izq).valor \leftarrow aux$ ▷ O(1)
    $z \leftarrow z.izq$ ▷ O(1)
  else
    $aux \leftarrow z.valor$ ▷ O(1)
    $z.valor \leftarrow (z.der).valor$ ▷ O(1)
    $(z.der).valor \leftarrow aux$ ▷ O(1)
    $z \leftarrow z.der$ ▷ O(1)
  end if
end function

4. Módulo Diccionario $_{TRIE}$ (α)

Interfaz

usa: .

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO).

Operaciones

DEFINIDA(**in** d: `dicc`$_{TRIE}$`(clave, significado)`, **in** c: `clave`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ def?(c,d)}
Complejidad: O(L)

OBTENER(**in** d: `dicc`$_{TRIE}$`(clave,significado)`, **in** c: `clave`) $\rightarrow$ res : `significado`
Pre $\equiv$ {def?(c,d)}
Post $\equiv$ {res $=_{obs}$ obtener(c,d)}
Complejidad: O(L)

VACIO() $\rightarrow$ res : `dicc`$_{TRIE}$`(clave,significado)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ vacio()}
Complejidad: O(1)

DEFINIR(**in/out** d: `dicc`$_{TRIE}$`(clave,significado)`, **in** c: clave, **in** s: `significado`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {$\neg$def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d=definir(c,d_0)}
Complejidad: O(L)

BORRAR(**in/out** d: `dicc`$_{TRIE}$`(clave,significado)`, **in** c: `clave`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {d=d_0}
Post $\equiv$ {d=borrar(c,d_0)}
Complejidad: O(L)
Descripción: Devuelve TRUE si se pudo borrar la clave, o FALSE si no la encontró

CLAVES(**in** d: `dicc`$_{TRIE}$`(clave,significado)`) $\rightarrow$ res : `conj(clave)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ claves(d)}
Complejidad: O()

5. Módulo Diccionario $_{AVL}$ (κ, σ)

Interfaz

usa: .

se explica con: DICCIONARIO(CLAVE,SIGNIFICADO).

géneros: $\texttt{dicc}_{AA}\texttt{(clave, significado)}$.

Operaciones

DEFINIDO?(**in** d: $\texttt{dicc}_{AA}\texttt{(clave, significado)}$, **in** c: `clave`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {}
Post $\equiv$ {res $=_{\text{obs}}$ def?(c,d)}
Complejidad: O(log(n))

SIGNIFICADO(**in** d: $\texttt{dicc}_{AA}\texttt{(clave,significado)}$, **in** c: `clave`) $\rightarrow$ res : `significado`
Pre $\equiv$ {def?(c,d)}
Post $\equiv$ {res $=_{\text{obs}}$ obtener(c,d)}
Complejidad: O(log(n))

VACIO() $\rightarrow$ res : $\texttt{dicc}_{AA}\texttt{(clave,significado)}$
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ vacio()}
Complejidad: O(1)

DEFINIR(**in/out** d: $\texttt{dicc}_{AA}\texttt{(clave,significado)}$, **in** c: `clave`, **in** s: `significado`)
Pre $\equiv$ {$\neg$def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d=definir(c,d_0)}
Complejidad: O()

BORRAR(**in/out** d: $\texttt{dicc}_{AA}\texttt{(clave,significado)}$, **in** c: `clave`)
Pre $\equiv$ {def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d=borrar(c,d_0)}
Complejidad: O()

Representación

$\texttt{dicc}_{AA}$ **se representa con** $\texttt{estr\_}AA$

donde $\texttt{estr}_{AA}$ es `tupla`(*id*: `nat`
izquierdo: `puntero(nodo(id,it))`
derecho: `puntero(nodo(id,it))`
valor: `itConj`(α)
)

Rep : estr $\longrightarrow$ bool

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

Abs : estr r $\longrightarrow$ dicc_{AA}(id, it) {Rep(r)}

Abs(r) $\equiv$ ...

Algoritmos

Algorithm 26 Implementación de Definido?

```
function iDefinido?(in d: estr_AA , in c: clave)→ res: bool
    nodoActual ← d
    res ← FALSE
    while ¬(nodoActual == NULO) && ¬res do
        if nodoActual.id == c then res ← TRUE
        else
            if c <nodoActual.id then nodoActual ← nodoActual.izquierdo
            elsenodoActual ← nodoActual.derecho
            end if
        end if
    end while
end function
```

Algorithm 27 Implementación de Significado

```
function iSignificado(in d: estr_AA , in c: clave)→ res: α
    nodoActual ← d
    res ← FALSE
    while ¬(nodoActual == NULO) && ¬res do
        if nodoActual.id == c then res ← nodoActual.valor
        else
            if c <nodoActual.id then nodoActual ← nodoActual.izquierdo
            elsenodoActual ← nodoActual.derecho
            end if
        end if
    end while
end function
```

Algorithm 28 Implementación de Vacio

```
function iVacio→ res: estr_AA
    res ← tupla(0, NULO, NULO, NULO, 0)
end function
```

Algorithm 29 Implementación de Definir

```
function iDefinir(in d: estr_AA , in c: nat, in s: α)
    nodoActual ← d
    yaDefini ← FALSE
    while ¬yaDefini do
        if c <nodoActual.id then
            if nodoActual.izquierdo == NULO then
                nuevoNodo ← Vacio()                                  ▷ Dicc_AA
                nuevoNodo.valor ← Copiar(s)
                nuevoNodo.id ← c
                nodoActual.izquierdo ← nuevoNodo
                yaDefini gets TRUE
            else
                nodoActual ← nodoActual.izquierdo
            end if
        else
            if nodoActual.derecho == NULO then
                nuevoNodo ← Vacio()                                  ▷ Dicc_AA
                nuevoNodo.valor ← Copiar(s)
                nuevoNodo.id ← c
                nodoActual.derecho ← nuevoNodo
                yaDefini ← TRUE
            else
                nodoActual ← nodoActual.derecho
            end if
        end if
    end while
    nodoActual ← Torsion(nodoActual)
    nodoACtual ← Division(nodoActual)
end function
```

Algorithm 30 Implementación de Torsion

```
function iTorsion(in d: estr_AA)→ res: estr_AA
    if d == NULO || d.izquierdo == NULO then
        res ← d
    else
        if Altura(d.izquierdo) ≥ Altura(d) then
            nodoAux ← d.izquierdo
            d.izquierdo ← nodoAux.derecho
            nodo.derecho ← d
            res ← nodoAux
        else
            res ← d
        end if
    end if
end function
```

Algorithm 31 Implementación de Division

```
function iDivision(in d: estr_AA)→ res: estr_AA
    if d == NULO || d.derecho == NULO || d.derecho.derecho == NULO then
        res ← d
    else
        if Altura(d.derecho.derecho) ≥ Altura(d) then
            nodoAux ← d.derecho
            d.derecho ← nodoAux.izquierdo
            nodoAux.izquierdo ← d
            nodoAux.altura ← nodoAux.altura + 1
            res ← nodoAux
        else
            res ← d
        end if
    end if
end function
```

Algorithm 32 Implementación de Altura

```
function iAltura(in d: estr_AA)→ res: nat_0
    if d.derecho == NULO && d.izquierdo == NULO then
        res ← 1
    else
        if d.derecho == NULO then
            res ← Altura(d.izquierdo) + 1
        else
            if d.izquierdo == NULO then
                res ← Altura(d.derecho) + 1
            else
                Minimo(Altura(d.izquierdo), Altura(d.derecho))          ▷ el mínimo de dos nat
            end if
        end if
    end if
end function
```