DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 2

Algoritmos y Estructuras de Datos II

Grupo: 4

Integrante	LU	Correo electrónico
Noli Villar, Juan Ignacio	174/14	juaninv@outlook.com
Langberg, Andrés	249/14	andreslangberg@gmail.com
Lew, Axel	225/14	axel.lew@hotmail.com
Cadaval, Matías Ezequiel	345/14	matias.cadaval@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1 Diseño del Tipo Red

1.1 Especificación

Se usa el Tad Red especificado por la cátedra.

1.2 Aspectos de la interfaz

1.2.1 Interfaz

Se explica con especificación de Red

Género `red`

Operaciones básicas de Red

MostrarComputadoras(**in** r: red) $\longrightarrow res : conj(pc)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $alias(res =_{\text{obs}} computadoras(r))$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de computadoras.*
Descripción: Devuelve un conjunto con las computadoras de la red.
Aliasing: res no es modificable.

EstanConectadas?(**in** r: red, **in** c_1: pc, **in** c_2: pc) $\longrightarrow res : bool$
Pre $\equiv$ { $c_1 \in computadoras(r) \land c_2 \in computadoras(r)$ }
Post $\equiv$ { $res =_{\text{obs}} conectadas?(r, c_1, c_2)$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de computadoras.*
Descripción: Devuelve *true* si y solo si c_1 y c_2 estan conectadas.

InterfazQueUsan(**in** r: red, **in** c_1: pc, **in** c_2:) $\longrightarrow res : interfaz$
Pre $\equiv$ { $c_1 \in computadoras(r) \land c_2 \in computadoras(r) \land_{\text{L}} conectadas?(r, c_1, c_2)$ }
Post $\equiv$ { $res =_{\text{obs}} interfazUsada(r, c_1, c_2)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la interfaz de c_1 por la cual las computadoras estan conectadas.

ArrancarRed() $\longrightarrow res : red$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} iniciarRed()$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Genera una red vacía

AgregarCompu(**in/out** r: red, **in** c: pc)
Pre $\equiv$ { $r =_{\text{obs}} r_0 \land (\forall c' : pc)\ c' \in computadoras(r) \ \Rightarrow\ ip(c) \neq ip(c')$ }
Post $\equiv$ { $r =_{\text{obs}} agregarComputadora(r_0)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Agrega la computadora c a la red.

Conectar(**in/out** r: red, **in** c_1: pc, **in** i_1: $interfaz$, **in** c_2: pc, **in** i_2: $interfaz$)
Pre $\equiv$ { $c_1 \in$ computadoras(r) $\land$ $c_2 \in$ computadoras(r) $\land$
$\pi_1(c_1) \neq \pi_1(c_2) \land \neg$conectadas?(r,$c_1, c_2$) $\land \neg$usaInterfaz?(r,c_1, i_1) $\land \neg$usaInterfaz?(r,c_2, i_2) $\land$ r = r_0 }
Post $\equiv$ { $r =_{\text{obs}} conectar(r_0, c_1, i_1, c_2, i_2)$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de computadoras.*
Descripción: Agrega una nueva conexión a la red.

ConectadoCon(**in** r: red, **in** c: pc) $\longrightarrow conj(pc)$
Pre $\equiv$ { $c \in computadoras(r)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} vecinos(r, c))$ }
Complejidad: $\mathcal{O}(n^2)$ n es la cantidad de computadoras.
Descripción: Devuelve el conjunto de computadoras con las cuales c esta conectada.
Aliasing: res no es modificable.

InterfazUsada?(**in** r: red, **in** c: pc, **in** i: $interfaz$) $\longrightarrow bool$
Pre $\equiv$ { $c \in computadoras(r)$ }
Post $\equiv$ { $res =_{\text{obs}} usaInterfaz?(r, c, i)$ }
Complejidad: $\mathcal{O}(c)$ *c es la cantidad de conexiones.*
Descripción: Devuelve true si y solo si la computadora usa la interfaz ingresada.

CaminosMinimos(**in** r: red, **in** c_1: pc, **in** c_2: pc) $\longrightarrow conj(lista(pc))$
Pre $\equiv$ { $c_1 \in computadoras(r) \land c_2 \in computadoras(r)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} caminosMinimos(r, c_1, c_2))$ }
Complejidad: $\mathcal{O}(n!)$
Descripción: Devuelve todos los caminos minimos existentes desde la pc c_1 hasta la pc c_2.
Aliasing: res no es modificable.

ExisteCamino?(**in** r: red, **in** c_1: pc, **in** c_2: pc) $\longrightarrow bool$
Pre $\equiv$ { $c_1 \in computadoras(r) \land c_2 \in computadoras(r)$ }
Post $\equiv$ { $res =_{\text{obs}} hayCamino?(r, c_1, c_2)$ }
Complejidad: $\mathcal{O}(n!)$
Descripción: Devuelve true si y solo si existe al menos un camino entre c_1 y c_2.

1.3 Pautas de implementación

1.3.1 Estructura de representación

red **se representa con** $estr$
donde $estr$ **es**
tupla(
interfaces: ***dicc(IP; conj(interfaz))*** $\times$
vecinos: ***dicc(IP; conj(IP))*** $\times$
conexiones: ***lista (tupla***$< pc_1 : IP;\ INT_1 : interfaz;\ pc_2 : IP;\ INT_2 : interfaz >$***)***
)

1.3.2 Justificación

1.3.3 Invariante de Representación

Informal

1. Las claves de los diccionarios interfaces y vecinos son las mismas.
2. Las computadoras no pueden estar conectadas a si mismas.
3. Los elementos de la lista de conexiones cumplen lo siguiente.
   (a) El primer y tercer elemento pertenecen a las claves de vecinos e interfaces.
   (b) El segundo y cuarto elemento son interfaces que corresponden a sus respectivas computadoras.
   (c) Si el elemento existe es porque el primer elemento tiene como vecino al tercero y viceversa, osea en la definición de los vecinos del primero va a estar al tercero y viceversa.
4. Los elementos obtenidos de obtener la definición de cualquier elemento del diccionario vecinos deben pertener a las claves de vecinos e interfaces.

Formal

Rep : estr $\longrightarrow$ *boolean*
($\forall$ e : estr)
Rep(e) $\equiv$ ($true \iff$
(1) claves(e.interfaces) $\leq$ claves(e.vecinos) $\land$ claves(e.interfaces = claves(e.vecinos)) $\land_{\text{L}}$
(2) ($\forall$ i $\in$ claves(e.vecinos)) $\neg$(i$\in$ pbtener(i,e.vecinos)) $\land_{\text{L}}$

(3) ($\forall$ t:$< IP, interfaz, IP, interfaz >$ esta?(e.conexiones,t)(π_1(t) $\in$ claves(e.interfaces)) $\wedge$ π_3(t) $\in$ claves(e.interfaces)
$\wedge_{\text{L}}$ π_2(t) $\in$ $significado$(π_1,e.interfaces) $\wedge_{\text{L}}$ π_4(t) $\in$ significado(π_3,e.interfaces)) $\wedge_{\text{L}}$
(π_1(t) $\in$ significado (π_3(t), e.vecinos)) $\wedge_{\text{L}}$ (π :$_3$(t) $\in$ significado(π_1(t), e.vecinos))
(4) ($\forall$ i:IP $\in$ claves(e.vecinos))(($\forall$ j:IP $\in$ significado(i,e.vecinos))(j $\in$ claves(e.interfaces))))

1.3.4 Función de Abstracción

Abs : estr e $\longrightarrow$ red {Rep(e)}
($\forall$ e:estr) Abs(e) $=_{\text{obs}}$ r : red /

(1)($\forall$ $c \in$ computadoras(r))((π_1(c) $\in$ claves(e.vecinos)) $\wedge$ π_2(c)$\subseteq$ significado(π_1(c),e.interfaces))$\wedge$
($\forall$ i $\in$ claves(e.vecinos))(i $\in$ juntarIP(computadoras(r))) $\wedge$ (significado(i,e.interfaces) = $\pi_2(buscar(i, computadoras(r)))\wedge$
(2)$\big(\forall\, c_1 \in$ computadoras(r))($\forall\, c_2 \in$ computadoras(r)$\big)$ conectadas(r,c_1, c_2) $\Leftrightarrow$ ($\pi_1(c_1) \in$ significado ($\pi_1(c_2)$,e.vecinos))
$\wedge$ $\pi_1(c_2) \in$ significado($\pi_1(c_1)$,e.vecinos) $\wedge$
(3) $\big($($\forall$ $c_1 \in$ computadoras(r))$\wedge$($\forall$ $c_2 \in$ computadoras(r))$\big)$ conectadas(r,c_1,c_2) $\Rightarrow$
interfazUsada(r,c_1,c_2)=π_2(buscar2($\pi_1(c_1)$,$\pi_1(c_2)$,e.conexiones))

Funciones Auxiliares

juntarIP : conj(pc) c $\longrightarrow$ conj(IP)
juntarIP(c) $\equiv$ **if** $\emptyset$?(c) **then** 0 **else** Ag(π_1(dameUno(c)), juntarIP(sinUno(c))) **fi**

buscar : IP ip $\times$ conj(pc) c $\longrightarrow$ pc
buscar(ip,c) $\equiv$ **if** i = π_1(dameUno(c)) **then** dameUno(c) **else** buscar(i,sinUno(c)) **fi**

buscar2 : IP ip$_1$ $\times$ IP ip$_2$ $\times$ Lista l $\longrightarrow$ pc
buscar2(ip_1,ip_2,l) $\equiv$ **if** π_1(primero(l)) = ip_1 $\wedge$ π_3(primero(l)) = ip_2 **then**
π_2(primero(l))
else
buscar2(ip_1,ip_2,fin(l))
fi

1.3.5 Algoritmos

1: **function** iARRANCARRED( ) $\longrightarrow$ res : estr ▷ $\mathcal{O}(1)$
2: var $dicc(IP, conj(interfaz))$ interfaces $\leftarrow$ vacio() ▷ $\mathcal{O}(1)$
3: var $dicc(IP, conj(IP))$ vecinos $\leftarrow$ vacio() ▷ $\mathcal{O}(1)$
4: var $lista(< IP, interfaz, IP, interfaz >)$ conexiones $\leftarrow$ vacio() ▷ $\mathcal{O}(1)$
5: res $\leftarrow$ < interfaces, vecinos, conexiones> ▷ $\mathcal{O}(1)$
6: **end function**

```
1: function iMostrarComputadoras(in r: estr) ⟶ res : conj(pc)                                   ▷ O(n)
2:     var conj(pc) compus ← vacio()                                                            ▷ O(1)
3:     var itDicc(IP, conj(IP)) claves ← crearIt(r.vecinos)                                     ▷ O(1)
4:     while (haySiguiente(claves)) do                                                          ▷ O(n)
5:         var conj(interfaz) interfaces ← significado(r.interfaces, siguiente(claves))         ▷ O(1)
6:         var tupla < IP, conj(interfaz) > pc ← < siguiente(clave), interfaces>                ▷ O(1)
7:         agregar(compus, pc)                                                                  ▷ O(1)
8:     end while
9:     res ← compus                                                                             ▷ O(1)
10: end function
```

```
1: function iInterfazQueUsan(in r: estr, in c1: pc, in c2: pc) ⟶ res : interfaz                 ▷ O(c)
2:     var itlista(< IP, interfaz, IP, interfaz >) conexiones ← crearIt(r.conexiones)           ▷ O(1)
3:     while (c1.IP != siguiente(conexiones).pc1) do                                            ▷ O(c)
4:         avanzar(conexiones)                                                                  ▷ O(1)
5:     end while
6:     res ← siguiente(conexiones).INT1                                                         ▷ O(1)
7: end function
```

```
1: function iEstanConectadas(in r: estr, in c1: pc, in c2: pc) ⟶ res : bool                     ▷ O(n)
2:     res ← pertenece?(significado(r.vecinos, c1.IP), c2)                                      ▷ O(n)
3: end function
```

```
1: function iAgregarCompu(in/out r: estr, in c: pc)                                             ▷ O(1)
2:     definir(r.interfaces, c.IP, c.interfaces)                                                ▷ O(1)
3:     var conj(IP) vecinos ← vacio()                                                           ▷ O(1)
4:     definir(r.vecinos, c.IP, vecinos)                                                        ▷ O(1)
5: end function
```

```
1: function iConectar(in/out r: estr, in c1: pc, in i1: interfaz, in c2: pc, in i2: interfaz)   ▷ O(n)
2:     var itDicc(IP, conj(IP)) IP1← crearIt(r.vecinos)                                         ▷ O(1)
3:     var itDicc(IP, conj(IP)) IP2← crearIt(r.vecinos)                                         ▷ O(1)
4:     while (c1.IP != siguiente(IP1) do                                                        ▷ O(n)
5:         avanzar(IP1)                                                                         ▷ O(1)
6:     end while
7:     while (c2.IP != siguiente(IP2) do                                                        ▷ O(n)
8:         avanzar(IP2)                                                                         ▷ O(1)
9:     end while
10:    agregar(siguienteSignificado(IP1), c2.IP)                                                ▷ O(1)
11:    agregar(siguienteSignificado(IP2), c1.IP)                                                ▷ O(1)
12:    var tupla < IP, interfaz, IP, interfaz > conexion ← < c1.IP, i1, c2.IP, i2¿              ▷ O(1)
13:    agregarAtras(r.conexiones, conexion)                                                     ▷ O(1)
14: end function
```

```
1: function iConectadoCon(in r: estr, in c: pc)⟶ res : conj(pc)                                   ▷ O(n^2)
2:     var conj(IP) vecinos ← significado(r.vecinos, c.IP)                                        ▷ O(n)
3:     var itConj(IP) vec ← crearIt(vecinos)                                                      ▷ O(1)
4:     var conj(pc) resu ← vacio()                                                                ▷ O(1)
5:     while (haySiguiente(vec)) do                                                               ▷ O(n)
6:         var itDicc(IP, conj(interfaz)) it ← crearIt(r.interfaces)                              ▷ O(1)
7:         while (siguiente(vec) != siguiente(it)) do                                             ▷ O(n)
8:             avanzar(it)                                                                        ▷ O(1)
9:         end while
10:        var tupla < pc, conj(interfaz) > compu ← < siguiente(vec), siguienteSignificado(it) >  ▷ O(1)
11:        eliminarSiguiente(vec)                                                                 ▷ O(1)
12:        avanzar(vec)                                                                           ▷ O(1)
13:        agregar(resu, compu)                                                                   ▷ O(1)
14:    end while
15:    res ← resu                                                                                 ▷ O(1)
16: end function
```

```
1: function iInterfazUsada(in r: estr, in c_1: pc, in i: interfaz)⟶ res : bool                    ▷ O(c)
2:     var itLista(tupla < IP, interfaz, IP, interfaz > conexion ← crearIt(r.conexiones)          ▷ O(1)
3:     var bool resu ← false                                                                      ▷ O(1)
4:     while (haySiguiente(conexion)) do                                                          ▷ O(c)
5:         if (siguiente(conexion).INT_1 = i) then                                                ▷ O(1)
6:             resu ← true                                                                        ▷ O(1)
7:         else
8:             avanzar(conexion)                                                                  ▷ O(1)
9:         end if
10:    end while
11:    res ← resu                                                                                 ▷ O(1)
12: end function
```

```
1: function iCaminos(in r: estr, in c_1: pc, in c_2: pc)⟶ res : conj(lista(IP))                   ▷ O(n!)
2:     var lista(IP) recorrido ← vacio()                                                          ▷ O(1)
3:     agregarAdelante(recorrido, c_1.IP)                                                         ▷ O(1)
4:     var conj(IP) candidatos ← significado(r.vecinos, c_1.IP)                                   ▷ O(n)
5:     res ← auxCaminos(r, c_1, c_2, recorrido, candidatos)                                       ▷ O(n!)
6: end function
```

```
1: function iRestaurar(in c: conj(lista(IP)), in r: estr)⟶ res : lista(pc) ▷ O(#(c) * n)c es el conjunto de entrada.
2:     var itConj(lista(IP)) it ← crearIt(c)                                                      ▷ O(1)
3:     while (haySiguiente(it)) do                                                                ▷ O(#(c))
4:         var itDicc(IP, conj(interfaz)) interfaces ← crearIt(r.interfaces)                      ▷ O(1)
5:         var itLista(IP) list ← crearIt(siguiente(it))                                          ▷ O(1)
6:         while (siguiente(list) != siguiente(interfaces)) do                                    ▷ O(n)
7:             avanzar(interfaces)                                                                ▷ O(1)
8:         end while
9:         var tupla < lista(IP), conj(interfaz) > compu ← <siguiente(it), siguienteSignificado(interfaces)>  ▷ O(1)
10:        avanzar(it)                                                                            ▷ O(1)
11:        agregar(res, compu)                                                                    ▷ O(1)
12:    end while
13: end function
```

```
1: function iAuxCaminos(in r: estr, in c1: pc, in c2: pc, in recorrido: lista(IP), in candidatos: conj(IP))⟶ res :
   conj(lista(IP))                                                                          ▷ O(n!)
2:     var conj(lista(IP)) vacio ← vacio()                                                  ▷ O(1)
3:     if (esVacio?(candidatos)) then                                                       ▷ O(1)
4:         res← vacio()                                                                     ▷ O(1)
5:     else
6:         if (ultimo=c2.IP) then                                                           ▷ O(1)
7:             res ← agregar(vacio, recorrido)                                              ▷ O(1)
8:         else
9:             if (¬ pertenece?(dameUno(candidatos),recorrido) then                         ▷ O(1)
10:                res ← union(auxCaminos(r,c1, c2,agregarAtras(recorrido,
11:                dameUno(candidatos)), significado(r.vecinos,dameUno(candidatos))),
12:                auxCaminos(r,c1, c2,recorrido,sinUno(candidatos)))                       ▷ O(n!)
13:            else
14:                res ←auxCaminos(r,c1, c2,recorrido,sinUno(candidatos))                   ▷ O(1)
15:            end if
16:        end if
17:    end if
18: end function
```

```
1: function iCaminosMinimos(in r: estr, in c1: pc, in c2: pc)⟶ res : conj(lista(pc))        ▷ O(n!)
2:     res ← restaurar(auxMinimos(caminos(r,c1, c2)), r)                                    ▷ O(n!)
3: end function
```

```
1: function iAuxMinimos(in cc: conj(lista(pc)))⟶ res : conj(lista(IP))                      ▷ O(n)
2:     var conjunto(lista(pc)) vacio ← vacio()                                              ▷ O(1)
3:     if (esVacio?(cc)) then                                                               ▷ O(1)
4:         res ← vacio()                                                                    ▷ O(1)
5:     else
6:         if (cardinal(cc) = 1) then                                                       ▷ O(1)
7:             agregar(vacio, dameUno(cc))                                                  ▷ O(1)
8:         else
9:             if (longitud(dameUno(cc)) < longitud(auxMinimos(sinUno(cc)))) then       ▷ O(auxMinimos − 1)
10:                agregar(vacio, dameUno(cc))                                              ▷ O(1)
11:            else
12:                if (longitud(dameUno(cc)) = longitud(dameUno(auxMinimos(sinUno(cc))))) then
13:                                                                                         ▷ O(auxMinimos − 1)
14:                    agregar(auxMinimos(sinUno(cc)), dameUno(cc))                         ▷ O(auxMinimos − 1)
15:                else
16:                    auxMinimos(sinUno(cc))                                               ▷ O(auxMinimos − 1)
17:                end if
18:            end if
19:        end if
20:    end if
21: end function
```

```
1: function iExisteCamino?(in r: estr, in c1: pc, in c2: pc)⟶ res : bool                    ▷ O(n!)
2:     res ← #(caminos(r,c1,c2)) ≥ 1                                                        ▷ O(n!)
3: end function
```

2 Diseño del Tipo DCNet

2.1 Especificación

Se usa el Tad DCNet especificado por la cátedra.

2.2 Aspectos de la interfaz

2.2.1 Interfaz

Se explica con especificación de DCNet

Género `dcnet`

Operaciones básicas de DCNet

Observacion: definimos la operacion $<$ para paquetes; un paquete es menor a otro cuando su id es menor

IniciarDCNet(**in** r: red) $\longrightarrow res : dcnet$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} iniciarDCNet(r)$ }
Complejidad: $\mathcal{O}(n! + n * L)$
Descripción: Devuelve un dcnet sin ningun paquete.

AñadirPaquete(**in/out** d: $dcnet$, **in** p: $paquete$)
Pre $\equiv$ { $d =_{\text{obs}} d_0 \wedge \neg \big((\exists\, p' : paquete)\ paqueteEnTransito?(d, p') \wedge$ id(p') $=_{\text{obs}}$ id(p)$\big) \wedge$ origen(p) $\in$ computadoras(red(d)) $\wedge$ destino(p) $\in$ computadoras(red(d)) $\wedge_{\text{L}}$ hayCamino?(d, origen(p), destino(p))}
Post $\equiv$ { $d =_{\text{obs}} crearPaquete(d_0, p)$ }
Complejidad: $\mathcal{O}(L + log(k))$
Descripción: Agrega al dcnet el paquete p.

AvanzarSegundo(**in/out** d: $dcnet$)
Pre $\equiv$ { $d =_{\text{obs}} d_0$ }
Post $\equiv$ { $d =_{\text{obs}} avanzarSegundo(d_0)$ }
Complejidad: $\mathcal{O}(n * (L + log(n) + log(k)))$
Descripción: Avanza un segundo en el dcnet y los paquetes de mayor prioridad pasan de una pc a otra siguiendo su camino correspondiente. Cada paquete puede moverse solo una vez, y cada pc puede enviar solo un paquete.

VerRed(**in** d: $dcnet$) $\longrightarrow res : red$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $alias(res =_{\text{obs}} red(d))$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la red que utiliza el dcnet.
Aliasing: res no es modificable.

Recorrido(**in** d: $dcnet$, **in** p: $paquete$) $\longrightarrow res : lista(pc)$
Pre $\equiv$ { $paqueteEnTransito?(d, p)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} caminoRecorrido(d, p))$ }
Complejidad: $\mathcal{O}(n * log(k))$
Descripción: Devuelve el camino recorrido por el paquete p desde que ingreso al dcnet.
Aliasing: res no es modificable.

Enviados(**in** d: $dcnet$, **in** c: pc) $\longrightarrow res : nat$
Pre $\equiv$ { $c \in$ computadoras(red(d))}

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

Complejidad: $\mathcal{O}(L)$

Descripción: Devuelve la cantidad de paquetes enviados por la pc c.

PAQUETES(**in** d: $dcnet$, **in** c: pc) $\longrightarrow res : conj(paquete)$
Pre $\equiv$ { $c \in$ computadoras(red(d))}
Post $\equiv$ { $alias(res =_{\text{obs}} enEspera(d, c))$ }

Complejidad: $\mathcal{O}(L)$

Descripción: Devuelve el conjunto de paquetes en espera que posee la pc c.

Aliasing: res no es modificable.

ENTRANSITO?(**in** d: $dcnet$, **in** p: $paquete$) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} paqueteEnTransito?(d, p)$ }

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

Descripción: Devuelve true si el paquete p pertenece al dcnet, y false en caso contrario.

MASENVIADOS(**in** d: $dcnet$) $\longrightarrow res : pc$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $alias(res =_{\text{obs}} laQueMasEnvio(d))$ }

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve la o una de las computadoras que mas paquetes envio (si es que hay mas de una) en el dcnet d.

Aliasing: res no es modificable.

2.3 Pautas de implementación

2.3.1 Estructura de representación

dcnet **se representa con** *estr*
donde *estr* **es**
tupla(
paquetes: ***diccTrie(pc; definicion)*** $\times$
masEnviados: ***tupla(pc:pc; cantidad:nat)*** $\times$
red: ***red*** $\times$
siguientes: ***diccAvl(pc; diccAvl(pc; pc))***
)
donde $definicion$ **es**
tupla(
xid: ***conjAvl(paquete)*** $\times$
xprior: ***colaHeap(paquete)*** $\times$
caminos: ***diccAvl(paquete, lista(pc))*** $\times$
enviados: ***nat***
)

2.3.2 Justificación

Para entender mejor la estructura damos una explicación:
Paquetes es un diccionario que dada una pc de la red nos devuelve una tupla *definicion* , donde *xid* es el conjunto de paquetes en espera que posee esa pc; *xprior* es una cola de prioridad de dichos paquetes; *caminos* es un diccionario que dado uno de los paquetes que tiene en espera la pc, nos devuelve el camino ya recorrido por ese paquete dentro del dcnet; y *enviados* es la cantidad de paquetes que envio la pc. *MasEnviados* es una tupla donde *pc* es la o una de las pcs que mas paquetes envio en el dcnet, y *cantidad* es el numero de paquetes que envio. *Red* es la red que utiliza el dcnet; y *siguientes* es un diccionario definido para todas las pcs del dcnet, que devuelve otro diccionario cuyas claves son únicamente las pcs del dcnet para las cuales existe un camino entre la clave del diccionario principal y esas pcs. La pc que devuelve el diccionario interno es la pc a la que deberia pasar un paquete si quisiera ir desde la pc clave de *siguientes* hasta la pc clave del diccionario interno para realizar el camino mas corto.

2.3.3 Invariante de Representación

Informal

1. Las claves de *paquetes* , las computadoras de *red* , y las claves de *siguientes* son el mismo conjunto.

2. La *pc* de *masEnviados* esta incluida en las computadoras de la *red* , exceptuando el caso en que la *red* no tenga ninguna pc, por lo tanto ese campo sera completado con algo arbitrario que no tendra importancia.

3. La *cantidad* de *masEnviados* es igual a los *enviados* de la *definicion* de la *pc* de *masEnviados* en el diccionario *paquetes* (si la *red* no contiene ni una pc entonces no estara definida ninguna clave).

4. Todas las *pc* definidas en *paquetes* tienen menor o igual *enviados* que la *pc* de *masEnviados* .

5. Para toda *pc* definida en *paquetes* , el conjunto *xid* , la cola *xprior* y el conjunto de claves de *caminos* , contienen exactamente los mismos elementos.

6. Todas las claves de los diccionarios que son significado de *siguientes* estan incluidas en el conjunto de claves de *paquetes* .

7. Para cada clave de *siguientes* , las claves del diccionario que es significado de dicha clave, cumplen que existe un camino en *red* entre cada una de ellas y la clave de *siguientes* .

8. Los significados del diccionario interno de *siguientes* son la segunda *pc* de uno de los caminos mínimos (si es que existe mas de uno) entre la clave principal y la clave del diccionario interno(obtenido a partir de la clave principal).

9. Cada una de las *lista*(*pc*) que son significado de *caminos* es prefijo de uno de los caminos mínimos (si es que existe mas de uno) entre la *pc* de origen y la *pc* destino de la clave (pues la clave es un paquete).

Formal

Rep : estr $\longrightarrow$ *boolean*

($\forall$ e : estr)

Rep(e) $\equiv$ ($true \iff$

(1) claves(e.paquetes) = claves(e.siguientes) = mostrarComputadoras(e.red) $\wedge$

(2) esVacio?(mostrarComputadoras(e.red)) $\vee$ (e.masEnviados).pc $\in$ claves(e.paquetes) $\wedge_{\text{L}}$

(3) esVacio?(mostrarComputadoras(e.red)) $\vee_{\text{L}}$

(e.masEnviados).cantidad = (obtener(e.paquetes,(e.masEnviados).pc)).enviados $\wedge$

(4) ($\forall$ c:claves(e.paquetes)) (obtener(e.paquetes, c)).enviados $\leq$ (e.masEnviados).cantidad $\wedge$

(5) ($\forall$ c:claves(e.paquetes)) $\Big($ (obtener(e.paquetes, c)).xid = claves((obtener(e.paquetes, c)).caminos) $\wedge$

($\forall$ p:paquete) p $\in$ (obtener(e.paquetes, c)).xid $\iff$ esta?((obtener(e.paquetes, c)).xprior, p) $\Big)$ $\wedge$

(6) ($\forall$ c:claves(e.paquetes)) claves(obtener(e.siguientes, c)) $\subseteq$ claves(e.paquetes) $\wedge$

(7) ($\forall$ c,d:claves(e.paquetes)) definido?(obtener(e.siguientes, c), d) $\iff$ existeCamino(e.red, c, d) $\wedge$

(8) ($\forall$ c:claves(e.paquetes))($\forall$ d:claves(obtener(e.siguientes, c))) $\wedge$

obtener(obtener(e.siguientes, c), d) = dameUno(caminosMasCortos(e.red,c, d))[1] $\wedge$

(9) ($\forall$ c:claves(e.paquetes))($\forall$ d:claves(obtener(e.siguientes, c))) $\wedge$

esPrefijo$\big($obtener(obtener(e.paquetes, c)).caminos, d), dameUno(caminosMasCortos(e.red, c, d))$\big)$)

Funciones Auxiliares

esta? : colaHeap(α) $\times$ α $\longrightarrow$ bool
esta?(c,a) $\equiv$ $\neg$vacia?(c) $\wedge_{\text{L}}$ $\big($proximo(c) = a $\vee$ esta?(desencolar(c), a)$\big)$

esPrefijo : lista(α) $\times$ lista(α) $\longrightarrow$ bool
esPrefijo(l,s) $\equiv$ longitud(l) $\leq$ longitud(s) $\wedge_{\text{L}}$ ($\forall$ i:[0..longitud(l))) l[i] = s[i]

2.3.4 Función de Abstracción

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

($\forall$ e:estr) Abs(e) $=_{\text{obs}}$ d : dcnet /

red(d) = e.red $\wedge_{\text{L}}$($\forall$ c:computadoras(red(d))) $\Big($cantidadEnviados(d, c) = (obtener(e.paquetes, c)).enviados $\wedge$

enEspera(d,c) = (obtener(e.paquetes, c)).xid $\wedge$ ($\forall$ p :claves(obtener(e.paquetes, c)))

caminoRecorrido(d, p) = obtener(obtener(e.paquetes, c), p) $\Big)$

2.3.5 Algoritmos

```
1: function iINICAR(in r: red) ⟶ res : estr                                   ▷ O(n! + n * L)
2:     var diccTrie(pc, definicion) paquetes ← vacio()                          ▷ O(1)
3:     var diccAvl(pc, diccAvl(pc, pc)) ipSiguiente ← vacio()                   ▷ O(1)
4:     if esVacio?(mostrarComputadoras(r)) then                                 ▷ O(1)
5:         var pc random ← <abc, vacio()>                                       ▷ O(1)
6:         var tupla < pc, nat > masEnviados ← <random, 0>                      ▷ O(1)
7:         var estr estructura ← <paquetes, masEnviados, r, ipSiguiente>        ▷ O(1)
8:     else
9:         var pc masEnvio ← dameUno(mostrarComputadoras(r))                    ▷ O(1)
10:        var tupla < pc, nat > masEnviados ← <masEnvio,0>                     ▷ O(1)
11:        var itConj(pc) computador1 ← crearIt(mostrarComputadoras(r))         ▷ O(1)
12:        while haySiguiente(computador1) do                                   ▷ O(n)
13:            var itConj(pc) computador2 ← crearIt(mostrarComputadoras(r))     ▷ O(1)
14:            var pc pc1 ← siguiente(computador1)                              ▷ O(1)
15:            var diccAvl(pc, pc) ipSiguiente2 ← vacio()                       ▷ O(1)
16:            while haySiguiente(computador2) do                               ▷ O(n)
17:                var pc pc2 ← siguiente(computador2)                          ▷ O(1)
18:                if ¬esVacio?(caminosMasCortos(r, pc1, pc2)) then             ▷ O(1)
19:                    definir(ipSiguiente2, pc2, dameUno(caminosMasCortos(r, pc1, pc2))[1])   ▷ O(n!)
20:                end if
21:                avanzar(computador2)                                         ▷ O(1)
22:            end while
23:            definir(ipSiguiente, pc1, ipSiguiente2)                          ▷ O(log(n))
24:            avanzar(computador1)                                             ▷ O(1)
25:        end while
26:        var itConj(pc) computador3 ← crearIt(mostrarComputadoras(r))         ▷ O(1)
27:        while haySiguiente(computador3) do                                   ▷ O(n)
28:            var conjAvl(paquete) paquetes2 ← vacio()                         ▷ O(1)
29:            var colaHeap(paquete) paquetes3 ← vacio()                        ▷ O(1)
30:            var nat paquetesEnviados ← 0                                     ▷ O(1)
31:            var diccAvl(paquete, lista(pc)) caminosRecorridos ← vacio()      ▷ O(1)
32:            definir(paquetes, siguente(computador3), <paquetes2, paquetes3, caminosRecorridos, paquetesEnviados>)
33:                                                                             ▷ O(L)
34:            avanzar(pc3)                                                     ▷ O(1)
35:        end while
36:        var estr estructura ← <paquetes, masEnviados, r, ipSiguiente>        ▷ O(1)
37:    end if
38:    res ← estructura                                                         ▷ O(1)
39: end function
```

```
1: function iAÑADIRPAQUETE(in/out dc: estr, in p: paquete)                     ▷ O(L + log(k))
2:     var puntero(tupla) actual ←  & obtener(dc.paquetes, p.origen)            ▷ O(L)
3:     agregar(actual→xid, p)                                                   ▷ O(log(k))
4:     encolar(actual→xprior, p)                                                ▷ O(log(k))
5:     var lista(pc) caminoRecorrido ← vacio()                                  ▷ O(1)
6:     agregarAtras(caminoRecorrido, p.origen)                                  ▷ O(1)
7:     definir(actual→caminos, p, caminoRecorrido)                              ▷ O(log(k))
8: end function
```

```
1: function iVERRED(in dc: estr) ⟶ res: red                                    ▷ O(1)
2:     res ← dc.red                                                             ▷ O(1)
3: end function
```

```
1: function iRecorrido(in dc: estr, in p: paquete) ⟶ res: lista(pc)                ▷ O(n * log(k))
2:     var bool noEncontrado ← true                                                 ▷ O(1)
3:     var itDiccTrie(pc, definicion) it ← crearIt(dc.paquetes)                     ▷ O(1)
4:     var puntero(definicion) actual                                               ▷ O(1)
5:     while haySiguiente(it) && noEncontrado do                                    ▷ O(n)
6:         actual ← siguienteSignificado(it)                                        ▷ O(1)
7:         if definido?(actual→caminos, p) then                                     ▷ O(log(k))
8:             noEncontrado ← false                                                 ▷ O(1)
9:             var lista(pc) camrec ← obtener(actual→caminos, p)                    ▷ O(log(k))
10:        else
11:            avanzar(it)                                                          ▷ O(1)
12:        end if
13:    end while
14:    res ← camrec                                                                 ▷ O(1)
15: end function
```

```
1: function iEnviados(in dc: estr, in c: pc) ⟶ res: nat                            ▷ O(L)
2:     res ← (obtener(dc.paquetes, c)).enviados                                     ▷ O(L)
3: end function
```

```
1: function iPaquetes(in dc: estr, in c: pc) ⟶ res: conjAvl(paquete)               ▷ O(L)
2:     res ← (obtener(dc.paquetes, c)).xid                                          ▷ O(L)
3: end function
```

```
1: function iMasEnviados(in dc: estr) ⟶ res: pc                                    ▷ O(1)
2:     res ← (dc.masEnviados).pc                                                    ▷ O(1)
3: end function
```

```
1: function iAvanzarSegundo(in/out dc: estr)                                   ▷ O(n * (L + log(n) + log(k)))
2:     var itConj(pc) it ← crearIt(mostrarComputadoras(dc.red))                ▷ O(1)
3:     var lista(tupla < paquete, lista(pc), pc >) aEnviar ← vacio()           ▷ O(1)
4:     while haySiguiente(it) do                                               ▷ O(n)
5:         var puntero(definicion) actual ← &obtener(dc.paquetes, siguiente(it))   ▷ O(L)
6:         if ¬vacio?(actual→xid) then                                         ▷ O(1)
7:             var paquete p ← desencolar(actual→xprior)                       ▷ O(log(k))
8:             var lista(pc) l ← obtener(actual→caminos, p)                    ▷ O(log(k))
9:             eliminar(actual→xid, p)                                         ▷ O(log(k))
10:            borrar(actual→caminos, p)                                       ▷ O(log(k))
11:            var puntero(diccAvl(pc, pc)) aux ← &obtener(dc.siguientes, siguiente(it))   ▷ O(log(n))
12:            var pc pct ← obtener(*aux, p.destino)                           ▷ O(log(n))
13:            if pct != p.destino then
14:                agregarAtras(l, pct)                                        ▷ O(1)
15:                agregarAtras(aEnviar, <p, l, pct>)                          ▷ O(1)
16:            end if
17:            actual→enviados ← actual→enviados + 1                           ▷ O(1)
18:            if (dc.masEnviados).pc < actual→enviados then                   ▷ O(1)
19:                dc.masEnviados ← <siguiente(it), actual→enviados>           ▷ O(1)
20:            end if
21:        end if
22:        avanzar(it)                                                         ▷ O(1)
23:    end while
24:    var itLista(tupla < paquete, lista(pc), pc >) itP ← crearIt(aEnviar)    ▷ O(1)
25:    while haySiguiente(itP) do                                              ▷ O(n)
26:        var puntero(definicion) actual2 ← &obtener(dc.paquetes, π3(siguiente(itP)))   ▷ O(L)
27:        agregar(actual2→xid, π1(siguiente(itP)))                            ▷ O(log(k))
28:        encolar(actual2→xprior, π1(siguiente(itP)))                         ▷ O(log(k))
29:        definir(actual2→caminos, π1(siguiente(itP)), π2(siguiente(itP)))    ▷ O(log(k))
30:        avanzar(itP)                                                        ▷ O(1)
31:    end while
32: end function
```

```
1: function iEnTransito?(in dc: estr, in p: paquete) ⟶ res: bool               ▷ O(n * (L + log(k)))
2:     var itConj(pc) it ← creatIt(mostrarComputadoras(dc.red))                ▷ O(1)
3:     var bool noEncontrado ← true                                            ▷ O(1)
4:     while noEncontrado && haySiguiente(it) do                               ▷ O(n)
5:         var puntero(definicion) sig ← &obtener(dc.paquetes, siguiente(it))  ▷ O(L)
6:         noEncontrado ← ¬(pertenece?(sig→xid, p))                            ▷ O(log(k))
7:         avanzar(it)                                                         ▷ O(1)
8:     end while
9:     res ← ¬noEncontrado                                                     ▷ O(1)
10: end function
```

3 Diseño del Tipo DiccionarioTrie(σ)

3.1 Especificación

Se usa el Tad Diccionario(κ, σ) especificado en el apunte de Tads básicos.

3.2 Aspectos de la interfaz

3.2.1 Interfaz

parámetros formales

género κ, σ

función $\bullet = \bullet$(**in** a_1: κ, **in** a_2: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} (a_1 = a_2)$ }
Complejidad: $\Theta(equals(a_1, a_2))$
Descripción: función de igualdad de κ's

función **COPIAR**(**in** k: κ) $\longrightarrow res : \kappa$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} k$ }
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

función **COPIAR**(**in** s: σ) $\longrightarrow res : \sigma$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} s$ }
Complejidad: $\Theta(copy(s))$
Descripción: función de copia de σ's

Se explica con especificación de Diccionario(κ, σ), Iterador Bidireccional(Tupla(κ, σ))

Género `diccTrie(`κ`,`σ`)`

Operaciones básicas de diccionario

Definido?(**in** d: $diccTrie(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} def?(d, k)$ }
Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*
Descripción: Devuelve true si y sólo si k está definido en el diccionario.

Obtener(**in** d: $diccTrie(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : \sigma$
Pre $\equiv$ { $def?(d, k)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} obtener(d, k))$ }
Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*
Descripción: Devuelve el significado de la clave k en d.
Aliasing: res no es modificable.

Vacio() $\longrightarrow res : diccTrie(\kappa, \sigma)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} vacio()$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Genera un diccionario vacío.

Definir(**in/out** d: $diccTrie(\kappa, \sigma)$, **in** k: κ, **in** s: σ)
Pre $\equiv$ { $d =_{\text{obs}} d_0$ }
Post $\equiv$ { $d =_{\text{obs}} definir(k, s, d_0)$ }
Complejidad: $\mathcal{O}(|k|)$ $|k|$ *es la longitud de la clave.*

Descripción: Define la clave k con el significado s en el diccionario.

CLAVES(**in** d: $diccTrie(\kappa, \sigma)$) $\longrightarrow res : conj(\kappa)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $alias(res =_{\text{obs}} claves(d))$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de claves.*
Descripción: Genera un conjunto con todas las claves del diccionario.
Aliasing: res no es modificable.

Operaciones básicas del iterador

CREARIT(**in** d: $diccTrie(\kappa, \sigma)$) $\longrightarrow res : itDiccTrie(\kappa, \sigma)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $alias(esPermutacion(SecuSuby(res), d)) \wedge vacia?(Anteriores(res))$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de claves.*
Descripción: Crea un iterador del diccionario de forma tal que se puedan recorrer sus elementos aplicando iterativamente SIGUIENTE(no ponemos la operacion SIGUIENTE en la interfaz pues no la usamos).

HAYSIGUIENTE(**in** it: $itDiccTrie(\kappa, \sigma)$) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} HaySiguiente?(it)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve true si y solo si en el iterador quedan elementos para avanzar.

SIGUIENTESIGNIFICADO(**in** it: $itDiccTrie(\kappa, \sigma)$) $\longrightarrow res : \sigma$
Pre $\equiv$ { $HaySiguiente?(it)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} Siguiente(it).significado)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el significado del elemento siguiente del iterador.
Aliasing: res no es modificable.

AVANZAR(**in/out** it: $itDiccTrie(\kappa, \sigma)$)
Pre $\equiv$ { $it =_{\text{obs}} it_0 \wedge$ HaySiguiente?(it)}
Post $\equiv$ { $it =_{\text{obs}} Avanzar(it_0)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Avanza a la posicion siguiente del iterador.

4 Diseño del Tipo Diccionario Avl(κ, σ)

4.1 Especificación

Se usa el Tad Diccionario(κ, σ) especificado en el apunte de Tads básicos.

4.2 Aspectos de la interfaz

4.2.1 Interfaz

parámetros formales

género κ, σ

función $\bullet = \bullet$(**in** a_1: κ, **in** a_2: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} (a_1 = a_2)$ }
Complejidad: $\Theta(equals(a_1, a_2))$
Descripción: función de igualdad de κ's

función **COPIAR**(**in** k: κ) $\longrightarrow res : \kappa$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} k$ }
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

función **COPIAR**(**in** s: σ) $\longrightarrow res : \sigma$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} s$ }
Complejidad: $\Theta(copy(s))$
Descripción: función de copia de σ's

Se explica con especificación de Diccionario(κ, σ), Iterador Bidireccional(Tupla(κ, σ))

Género `diccAvl(`κ, σ`)`

Operaciones básicas de diccionario

Definido?(**in** d: $diccAvl(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} def?(d, k)$ }
Complejidad: $\mathcal{O}(log(n))$ *n es la cantidad de claves.*
Descripción: Devuelve true si y sólo si k está definido en el diccionario.

Obtener(**in** d: $diccAvl(\kappa, \sigma)$, **in** k: κ) $\longrightarrow res : \sigma$
Pre $\equiv$ { $def?(d, k)$ }
Post $\equiv$ { $alias(res =_{\text{obs}} obtener(d, k))$ }
Complejidad: $\mathcal{O}(log(n))$ *n es la cantidad de claves.*
Descripción: Devuelve el puntero al significado de la clave k en d.
Aliasing: res no es modificable.

Vacio() $\longrightarrow res : diccAvl(\kappa, \sigma)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} vacio()$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Genera un diccionario vacío.

Definir(**in/out** d: $diccAvl(\kappa, \sigma)$, **in** k: κ, **in** s: σ)
Pre $\equiv$ { $d =_{\text{obs}} d_0$ }
Post $\equiv$ { $d =_{\text{obs}} definir(k, s, d_0)$ }
Complejidad: $\mathcal{O}(log(n))$ *n es la cantidad de claves.*

Descripción: Define la clave k con el significado s en el diccionario.

BORRAR(**in/out** d: $diccAvl(\kappa, \sigma)$, kκ)
Pre $\equiv$ { $d =_{\text{obs}} d_0 \wedge$ def?(d,k)}
Post $\equiv$ { $d =_{\text{obs}} borrar(d_0, k)$ }
Complejidad: $\mathcal{O}(log(n))$ *n es la cantidad de claves.*
Descripción: Borra del diccionario la clave k y su significado.

CLAVES(**in** d: $diccAvl(\kappa, \sigma)$) $\longrightarrow res : conj(\kappa)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $alias(res =_{\text{obs}} claves(d)$ }
Complejidad: $\mathcal{O}(n)$ *n es la cantidad de claves.*
Descripción: Genera un conjunto con todas las claves del diccionario.
Aliasing: res no es modificable.

4.3 Pautas de implementación

4.3.1 Estructura de representación

$diccAvl(\kappa, \sigma)$ **se representa con** $puntero(nodo)$
donde $nodo$ **es**
tupla(
significado: ***puntero(σ)*** ×
der: ***puntero(nodo)*** ×
izq: ***puntero(nodo)*** ×
padre: ***puntero(nodo)*** ×
key: κ ×
alt: ***nat***
)

4.3.2 Justificación

Para entender mejor la estructura damos una explicación:
Cada nodo respresenta un AVL, donde $significado$ contiene la definicion de la clave key; der e izq son el hijo derecho del nodo y el hijo izquierdo del nodo respectivamente; $padre$ contiene a su nodo padre ; y alt representa la altura del AVL.

4.3.3 Invariante de Representación

Informal

1. La clave de cada nodo es mayor que la clave de su hijo derecho y menor que la de su hijo izquierdo.
2. La altura de los subarboles izq y der difieren a lo sumo en 1.
3. Todos los nodos del subarbol izq/der son menores/mayores que el nodo raiz.
4. No hay 2 punteros al mismo nodo ni ciclos.
5. El rep se cumple para todos los subarboles del AVL.

Formal

Rep : estr $\longrightarrow$ *boolean*
($\forall$ e : estr)
Rep(e) $\equiv$ ($true \iff$
(1) e = NULL $\vee_{\text{L}}$ $\big($(e$\rightarrow$ izq $\neq$ NULL) $\Rightarrow_{\text{L}}$ e $\rightarrow$ izq $\rightarrow$ key $<$ e $\rightarrow$ key $\wedge$
(e$\rightarrow$ der $\neq$ NULL) $\Rightarrow_{\text{L}}$ e $\rightarrow$ der $\rightarrow$ key $>$ e $\rightarrow$ key $\big)$ $\wedge$
(2) e = NULL $\vee_{\text{L}}$ $\big($(e $\rightarrow$ der = NULL) $\Rightarrow$ e $\rightarrow$ izq $\rightarrow$ alt ≤ 1 $\wedge$
(e $\rightarrow$ izq = NULL) $\Rightarrow$ e $\rightarrow$ der $\rightarrow$ alt ≤ 1 $\wedge$
(e$\rightarrow$ der $\neq$ NULL $\wedge$ e $\rightarrow$ izq $\neq$ NULL) $\Rightarrow$

(maxAlt(e→ der, e→ izq))→ alt - (minAlt(e→ der, e→ izq))→ alt ≤ 1) ∧

(3) e ≠ NULL ⇒$_{L}$ (∀ n:nodo) (n ∈ hijos(*(e→ der)) ⇒ e→ key < n.key ∧
n ∈ hijos(*(e→ izq)) ⇒ e→ key > n.key)

(4) e = NULL ∨$_{L}$ (∀ n,m: nodo) n ∈ hijos(*e) ∧ m ∈ hijos(*e) ⇒ apuntanDistinto(n, m) ∧ nadiePadreDeRaiz(n) ∧$_{L}$ sonPadreEHijo(n,m)

(5) Rep(e → der) ∧ Rep(e → izq))

4.3.4 Función de Abstracción

Abs : estr e ⟶ diccAvl (κ, σ) {Rep(e)}

(∀ e:estr) Abs(e) =$_{obs}$ d : diccAvl (κ, σ) /

If e = NULL **then** d = vacio() **else** d = definir(agregar(abs(e → der), abs(e → izq)),e → key, e → significado)

Funciones Auxiliares

agregar : diccAvl($\kappa \times \sigma$) × diccAvl($\kappa \times \sigma$) ⟶ diccAvl(κ, σ)

agregar(a,b) ≡ **if** #claves(a) == 0 **then**
b
else
definir(dameUno(claves(a)), obtener(dameUno(claves(a)),a),
agregar(borrar(dameUno(claves(a)),a),b))
fi

maxAlt : puntero(nodo) × puntero(nodo) ⟶ puntero(nodo)

maxAlt(n,m) ≡ **if** n = NULL ∧ m = NULL **then** n **else**
(**if** n = NULL **then** m **else**
(**if** m = NULL **then** n **else**
(**if** n→ alt > m→ alt **then** n **else** m **fi**) **fi**) **fi**) **fi**

minAlt : puntero(nodo) × puntero(nodo) ⟶ puntero(nodo)

minAlt(n,m) ≡ **if** n = NULL ∧ m = NULL **then** n **else**
(**if** n = NULL **then** m **else**
(**if** m = NULL **then** n **else**
(**if** n→ alt < m→ alt **then** n **else** m **fi**) **fi**) **fi**) **fi**

Explicamos el funcionamiento de *apuntanDistinto* ; *nadiePadreDeRaiz* y *sonPadreEHijo*:

Dados 2 nodos *apuntanDistinto* devuelve true si y solo si los hijos(izq y der) de dichos nodos no apuntan a mismas posiciones de memoria, exceptuando que sean NULL.

Dado un nodo *nadiePadreDeRaiz* devuelve true si y solo si los hijos(izq y der) de dicho nodo no apuntan a la raiz del AVL.

Dados 2 nodos $sonPadreEHijo(n, m)$ devuelve true si y solo si, uno de los nodos tiene como hijo al otro ⟺ ese otro tiene como padre al primer nodo.

4.3.5 Algoritmos

```
1: function iVacio( ) ⟶ res : estr                                   ▷ O(1)
2:     res ← NULL                                                    ▷ O(1)
3: end function
```

```
1: function iDefinir(in/out d: estr, in k: κ, in s: σ)               ▷ O(log2(k))
2:     if d = NULL then
3:         var puntero(nodo) nuevoNodo                               ▷ O(1)
4:         var nodo aux ← < s, NULL, NULL, NULL, 1, k >              ▷ O(1)
5:         *nuevoNodo ← aux                                          ▷ O(1)
6:         d→raiz ← nuevoNodo                                        ▷ O(1)
7:     else
8:         var puntero(nodo) n ← d                                   ▷ O(1)
9:         var puntero(nodo) papa                                    ▷ O(1)
10:        while true do                                             ▷ O(log2(k))
11:            if k = n→key then                                     ▷ O(1)
12:                break                                             ▷ O(1)
13:            end if
14:            papa ← n                                              ▷ O(1)
15:            var bool porIzq? ← n→key > k                          ▷ O(1)
16:            if porIzq? then                                       ▷ O(1)
17:                n = n→izq                                         ▷ O(1)
18:            else
19:                n = n→der                                         ▷ O(1)
20:            end if
21:            if n = NULL then                                      ▷ O(1)
22:                if porIzq? then                                   ▷ O(1)
23:                    papa→izq ← < s, NULL, NULL, papa, 1, k >      ▷ O(1)
24:                else
25:                    papa→der ← < s, NULL, NULL, papa, 1, k >      ▷ O(1)
26:                end if
27:                rebalanceo(d,papa)                                ▷ O(log2(k))
28:                break                                             ▷ O(1)
29:            end if
30:        end while
31:    end if
32: end function
```

```
1: function iBorrar(in/out d: estr, in k: α)                          ▷ O(log2(k))
2:     if d != NULL then                                               ▷ O(1)
3:         var puntero(nodo) n ← d                                     ▷ O(1)
4:         var puntero(nodo) papa ← d                                  ▷ O(1)
5:         var puntero(nodo) bNodo ← NULL                              ▷ O(1)
6:         var puntero(nodo) hijo ← d                                  ▷ O(1)
7:         while hijo != NULL do                                       ▷ O(log2(k))
8:             papa ← n                                                ▷ O(1)
9:             n ← hijo                                                ▷ O(1)
10:            if k ≥ n→key then                                       ▷ O(1)
11:                n ← n→der                                           ▷ O(1)
12:            else
13:                n ← n→izq                                           ▷ O(1)
14:            end if
15:            if k = n→key then                                       ▷ O(1)
16:                bNodo ← n                                           ▷ O(1)
17:            end if
18:        end while
19:        if bNodo != NULL then                                       ▷ O(1)
20:            bNodo→key ← n→key                                       ▷ O(1)
21:            if n→izq != NULL then                                   ▷ O(1)
22:                hijo ← n→izq                                        ▷ O(1)
23:            else
24:                hijo ← n→der                                        ▷ O(1)
25:            end if
26:            if raiz→key = k then                                    ▷ O(1)
27:                raiz ← hijo                                         ▷ O(1)
28:            else
29:                if papa→izq = n then                                ▷ O(1)
30:                    papa→izq ← hijo                                 ▷ O(1)
31:                else
32:                    papa→der ← hijo                                 ▷ O(1)
33:                end if
34:                rebalanceo(d,papa)                                  ▷ O(log2(k))
35:            end if
36:        end if
37:    end if
38: end function
```

```
1: function iRebalanceo(in/out d: estr in/out n: nodo)                          ▷ O(log2(k))
2:     balancear(n)                                                             ▷ O(1)
3:     var int balanceo ← (n→der→altura) - (n→izq→altura)                        ▷ O(1)
4:     if balanceo = -2 then                                                    ▷ O(1)
5:         if n→izq→izq→altura ≥ n→izq→der→altura then                          ▷ O(1)
6:             n ← rotacionDerecha(n)                                           ▷ O(1)
7:         else
8:             n ← rotacionIzqDer(n)                                            ▷ O(1)
9:         end if
10:    else
11:        if balanceo = 2 then                                                 ▷ O(1)
12:            if n→der→der→altura ≥ n→der→izq→altura then                      ▷ O(1)
13:                n ← rotacionIzquierda(n)                                     ▷ O(1)
14:            else
15:                n ← rotacionDerIzq(n)                                        ▷ O(1)
16:            end if
17:        end if
18:    end if
19:    if n→padre != NULL then                                                  ▷ O(1)
20:        rebalanceo(d,n→padre)                ▷ O(rebalanceod, n→padre)   else
21:        d ← n                                                                ▷ O(1)
23:    end if
24: end function
```

```
1: function iRotacionIzquierda(in/out a: Nodo) ⟶ res : Nodo                     ▷ O(1)
2:     var puntero(nodo) b ← a→der                                              ▷ O(1)
3:     b→padre ← a→padre                                                        ▷ O(1)
4:     a→der ← b→izq                                                            ▷ O(1)
5:     if a→der != NULL then                                                    ▷ O(1)
6:         a→der→padre ← a                                                      ▷ O(1)
7:     end if
8:     b→izq ← a                                                                ▷ O(1)
9:     a→padre ← b                                                              ▷ O(1)
10:    if b→padre != NULL then                                                  ▷ O(1)
11:        if b→padre→der = a then                                              ▷ O(1)
12:            b→padre→der ← b                                                  ▷ O(1)
13:        else
14:            b→padre→izq ← b                                                  ▷ O(1)
15:        end if
16:    end if
17:    balancear(a)                                                             ▷ O(1)
18:    balancear(b)                                                             ▷ O(1)
19:    res ← b                                                                  ▷ O(1)
20: end function
```

```
1: function iRotacionDerecha(in/out a: Nodo)⟶ res : Nodo                ▷ O(1)
2:     var puntero(nodo) b ← a→izq                                       ▷ O(1)
3:     b→padre ← a→padre                                                 ▷ O(1)
4:     a→izq ← b→der                                                     ▷ O(1)
5:     if a→izq != NULL then                                             ▷ O(1)
6:         a→izq→padre ← a                                               ▷ O(1)
7:     end if
8:     b→der ← a                                                         ▷ O(1)
9:     a→padre ← b                                                       ▷ O(1)
10:    if b→padre != NULL then                                           ▷ O(1)
11:        if b→padre→der = a then                                       ▷ O(1)
12:            b→padre→der ← b                                           ▷ O(1)
13:        else
14:            b→padre→izq ← b                                           ▷ O(1)
15:        end if
16:    end if
17:    balancear(a)                                                      ▷ O(1)
18:    balancear(b)                                                      ▷ O(1)
19:    res ← b                                                           ▷ O(1)
20: end function
```

```
1: function iRotacionIzqDer(in/out n: Nodo)⟶ res : Nodo                 ▷ O(1)
2:     n→izq ← rotacionIzquierda(n→izq)                                  ▷ O(1)
3:     res ← rotacionDerecha(n)                                          ▷ O(1)
4: end function
```

```
1: function iRotacionDerIzq(in/out n: Nodo)⟶ res : Nodo                 ▷ O(1)
2:     n→der ← rotacionDerecha(n→der)                                    ▷ O(1)
3:     res ← rotacionIzquierda(n)                                        ▷ O(1)
4: end function
```

```
1: function iBalancear(in/out n: Nodo)                                  ▷ O(1)
2:     n→altura ← (n→der→altura) + (n→izq→altura) + 1                    ▷ O(1)
3: end function
```

```
1: function iDefinido?(in n: estr, in k: κ)⟶ res : bool                    ▷ O(log2(k))
2:     if d = NULL then                                                     ▷ O(1)
3:         res ← false                                                      ▷ O(1)
4:     else
5:         var puntero(nodo) n ← d                                          ▷ O(1)
6:         while true do                                                    ▷ O(log2(k))
7:             if n→key = k then                                            ▷ O(1)
8:                 res ← true                                               ▷ O(1)
9:                 break                                                    ▷ O(1)
10:            end if
11:            if n→key > k then                                            ▷ O(1)
12:                n ← n→izq                                                ▷ O(1)
13:            else
14:                n ← n→der                                                ▷ O(1)
15:            end if
16:            if n = NULL then                                             ▷ O(1)
17:                res ← false                                              ▷ O(1)
18:                break                                                    ▷ O(1)
19:            end if
20:        end while
21:    end if
22: end function
```

```
1: function iSignificado(in n: estr, in k: κ)⟶res :σ                       ▷ O(log2(k))
2:     var puntero(nodo) n ← d                                              ▷ O(1)
3:     while true do                                                        ▷ O(log2(k))
4:         if n→key = k then                                                ▷ O(1)
5:             res ← n→significado                                          ▷ O(1)
6:             break                                                        ▷ O(1)
7:         end if
8:         if n→key > k then                                                ▷ O(1)
9:             n ← n→izq                                                    ▷ O(1)
10:        else
11:            n ← n→der                                                    ▷ O(1)
12:        end if
13:    end while
14:
15: end function
```

```
1: function iClaves(in/out d: estr)⟶ res : conj(κ)                          ▷ O(n)
2:     var conj(κ) c ← vacio()                                              ▷ O(1)
3:     if d != NULL then                                                    ▷ O(1)
4:         res ← unir(agregar(c, e→key), unir(claves(e→izq), claves(e→der)))  ▷ O(n)
5:     else
6:         res ← vacio()                                                    ▷ O(1)
7:     end if
8: end function
```

5 Diseño del Tipo ConjAvl(α)

5.1 Especificación

Se usa el Tad Conjunto(α) especificado por la cátedra.

5.2 Aspectos de la interfaz

5.2.1 Interfaz

Se explica con especificación de Conjunto(α)

Género `conjAvl(`α`)`

Operaciones básicas de ConjAvl(α)

Vacio() $\longrightarrow res : conjAvl(\alpha)$

Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} vacio()$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un conjAvl(α) vacio.

Agregar(**in/out** c: $conjAvl(\alpha)$, **in** a: α)

Pre $\equiv$ { $c =_{\text{obs}} c_0$ }
Post $\equiv$ { $res =_{\text{obs}} Ag(a, c_0)$ }
Complejidad: $\mathcal{O}(log(n))$
Descripción: Agrega el elemento a al conjAvl c.

Vacio?(**in** c: $conjAvl(\alpha)$) $\longrightarrow res : bool$

Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} \emptyset?(c)$ }
Complejidad: $\mathcal{O}()$
Descripción: Devuelve true si y solo si c esta vacio.

Pertenece?(**in** c: $conjAvl(\alpha)$, **in** a: α) $\longrightarrow res : bool$

Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} a \in c$ }
Complejidad: $\mathcal{O}(log(n))$
Descripción: Devuelve true si y solo si a pertenece al conjunto.

Eliminar(**in/out** c: $conjAvl(\alpha)$, **in** a: α)

Pre $\equiv$ { $c =_{\text{obs}} c_0$ }
Post $\equiv$ { $c =_{\text{obs}} c \backslash \{a\}$ }
Complejidad: $\mathcal{O}(log(n))$
Descripción: Elimina a de c si es que estaba.

5.3 Pautas de implementación

5.3.1 Estructura de representación

$conjAvl(\alpha)$ **se representa con** $estr$
donde $estr$ **es** diccAvl($\alpha, bool$)

5.3.2 Justificación

Para entender mejor la estructura damos una explicación:
Representamos conjAvl(α) mediante un diccAvl(α, bool) donde los elementos del conjunto van a ser las claves del diccionario, y elegimos arbitrariamente el tipo bool para el significado, pero no nos va a interesar.

5.3.3 Invariante de Representación

Informal

1. No necesitamos pedir nada en el invariante de representacion pues cualquier diccAvl(α, bool) nos sirve para representar algun conjunto.

Formal

Rep : estr $\longrightarrow$ *boolean*
($\forall$ e : estr)
Rep(e) $\equiv$ (true)

5.3.4 Función de Abstracción

Abs : estr e $\longrightarrow$ conjAvl(α) $\{Rep(\text{e})\}$
($\forall$ e:estr) Abs(e) $=_{\text{obs}}$ c : conjAvl(α)/
($\forall$ d:α) definido?(e, d) $\Longleftrightarrow$ d $\in$ c

5.3.5 Algoritmos

```
1: function iVacio⟶ res: estr                                   ▷ O(1)
2:     res ← vacio()                                            ▷ O(1)
3: end function
```

```
1: function iAgregar(in/out c: estr, in a: α)                   ▷ O(log(n))
2:     definir(c, a, true)                                      ▷ O(log(n))
3: end function
```

```
1: function iVacio?(in c: estr)⟶ res: bool                      ▷ O(1)
2:     res ← esVacio?(claves(c))                                ▷ O(1)
3: end function
```

```
1: function iEliminar(in/out c: estr, in a: α)                    ▷ O(log(n))
2:     if definido?(c, a) then                                    ▷ O(log(n))
3:         borrar(c, a)                                           ▷ O(log(n))
4:     end if
5: end function
```

```
1: function iPertence?(in c: estr, in a: α)⟶ res: bool            ▷ O(log(n))
2:     res ← definido?(c, a)                                      ▷ O(log(n))
3: end function
```

6 Diseño del Tipo ColaHeap(paquete)

6.1 Especificación

Se usa el Tad Cola de Prioridad(α) especificado en el apunte de Tads básicos.

6.2 Aspectos de la interfaz

6.2.1 Interfaz

parámetros formales

género paquete

función $\bullet = \bullet$(**in** a_1: $paquete$, **in** a_2: $paquete$) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} (a_1 = a_2)$ }
Complejidad: $\Theta(equals(a_1, a_2))$
Descripción: función de igualdad de paquetes

función **COPIAR**(**in** k: $paquete$) $\longrightarrow res : paquete$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} k$ }
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de paquetes

Se explica con especificación de Cola de Prioridad(α)

Género `colaHeap(paquete)`

Operaciones básicas de colaHeap

Vacia() $\longrightarrow res : colaHeap(paquete)$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} vacia()$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Genera una cola vacia.

Encolar(**in/out** c: $colaHeap(paquete)$, **in** a: $paquete$) $\longrightarrow res : colaHeap(paquete)$
Pre $\equiv$ { $c =_{\text{obs}} c_0$ }
Post $\equiv$ { $c =_{\text{obs}} encolar(a, c_0)$ }
Complejidad: $\mathcal{O}(log(n)$ *n es la cantidad de paquetes.*$)$
Descripción: Agrega el elemento a en la cola.
Aliasing:

Vacia?(**in** c: $colaHeap(paquete)$) $\longrightarrow res : bool$
Pre $\equiv$ { $true$ }
Post $\equiv$ { $res =_{\text{obs}} vacia?(c)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Comprueba si la cola esta vacia
Aliasing:

Desencolar(**in/out** c: $colaHeap(paquete)$) $\longrightarrow res : paquete$

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

Complejidad: $\mathcal{O}(log(n))$ *n es la cantidad de paquetes.*

Descripción: Elimina el proximo de la cola y devuelve el eliminado.

Aliasing:

6.3 Pautas de implementación

6.3.1 Estructura de representación

colaHeap(*paquete*) **se representa con** *estr*

donde *estr* **es**

tupla(
raiz: ***puntero(nodo)*** $\times$
padreUlt: ***puntero(nodo)*** $\times$
cant : ***nat***
)

donde *nodo* **es**

tupla(
izq: ***puntero(nodo)*** $\times$
der: ***puntero(nodo)*** $\times$
padre: ***puntero(nodo)*** $\times$
elem : ***paquete***
)

6.3.2 Invariante de Representación

Informal

1. La priroridad de cada nodo es mayor que la de sus hijos, es izquierdista, esta balanceado, si un nodo es hijo de un nodo N, su puntero padre apunta a N.

2. *Cant* es la cantidad de elementos.

3. *PadreUlt* apunta al ultimo (de izquerda a derecha) nodo incompleto de un nivel completo (completo es que el nivel tiene todos los nodos posibles).

4. No hay nodos repetidos ni ciclos.

Formal

Rep : estr $\longrightarrow$ *boolean*

($\forall$ e : estr)

Rep(e) $\equiv$ ($true \iff$

(1)esHeap?(c.raiz) $\wedge_{\text{L}}$

(2)c.cant $=_{\text{obs}}$ cantNodos(c.raiz) $\wedge_{\text{L}}$

(3)c.cant $> 1 \Rightarrow_{\text{L}}$c.padreUlt $=_{\text{obs}}$ buscarPadreUlt(c.raiz)

(4)($\forall p$: puntero(nodo)) (p $\in$ punteros(r.raiz)) $\Rightarrow$ #(p, punteros(r.raiz)) $=_{\text{obs}}$ 1))

6.3.3 Función de Abstracción

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

($\forall$ e:estr) Abs(e) $=_{\text{obs}}$ c : colaPrior(paquete) /

if c.cant = 0 **then** vacia **else** encolado(colaSecu(c.raiz)) **fi**

Funciones Auxiliares

esHeap? : puntero(nodo) n $\longrightarrow$ bool
esHeap?(n) $\equiv$ **if** n = NULL **then** true **else** $\neg$(n$\rightarrow$der = NULL) $\Rightarrow$ $\neg$(n$\rightarrow$izq = NULL) $\wedge$
if n$\rightarrow$izq = NULL **then** true **else** (n$\rightarrow$elem).prioridad > ★ $\wedge$ n$\rightarrow$izq$\rightarrow$padre = n $\wedge$
if n$\rightarrow$der = NULL **then**
true
else
(n$\rightarrow$elem).prioridad > (n$\rightarrow$der$\rightarrow$elem).prioridad $\wedge$ n$\rightarrow$der$\rightarrow$padre = n **fi** **fi**
$\wedge$ altura(n$\rightarrow$izq)-altura(n$\rightarrow$der) <= 1 $\wedge$ esHeap?(n$\rightarrow$izq) $\wedge$ esHeap?(n$\rightarrow$der)
fi

★==(n$\rightarrow$izq$\rightarrow$elem).prioridad

cantNodos : puntero(nodo) n $\longrightarrow$ nat
cantNodos(n) $\equiv$ **if** n = NULL **then** 0 **else** 1 + cantNodos(n$\rightarrow$izq) + cantNodos(n$\rightarrow$der) **fi**

altura : puntero(nodo) n $\longrightarrow$ nat
altura(n) $\equiv$ **if** n = NULL **then** 0 **else** 1 + max(cantNodos(n$\rightarrow$izq), cantNodos(n$\rightarrow$der)) **fi**

buscarPadreUlt : puntero(nodo) n $\longrightarrow$ nat {cantNodos(n) > 1 $\wedge$*esHeap?*(n)}
buscarPadreUlt(n) $\equiv$ **if** n$\rightarrow$der = NULL $\vee$*altura*($n$$\rightarrow$der) = 1 **then** n **else** **if** altura(n$\rightarrow$der) + 1 = altura(n$\rightarrow$izq)
then
buscarPadreUlt(n$\rightarrow$izq)
else
buscarPadreUlt(n$\rightarrow$der) **fi**
fi
punteros : puntero(nodo) n $\longrightarrow$ multiconjunto(puntero(nodo))
punteros(n) $\equiv$ **if** n = NULL **then** $\emptyset$ **else** Ag(n,punteros(n$\rightarrow$izq)) $\cup$ punteros(n$\rightarrow$der) **fi**

encolado : secu(paquete) s $\longrightarrow$ ColaPrior(paquete)
encolado(s) $\equiv$ **if** vacia?(s) **then** vacia **else** encolar(prim(s),encolado(fin(s))) **fi**

colaSecu : puntero(nodo) n $\longrightarrow$ secu(paquete)
colaSecu(n) $\equiv$ **if** n = NULL **then** $<>$ **else** n$\rightarrow$elem $\bullet$ colaSecu(n$\rightarrow$izq) colaSecu(n$\rightarrow$der) **fi**

6.3.4 Algoritmos

```
1: function iEncolar(in/out c: estr, in p: paquete)                          ▷ O(log2(k))
2:     var nodo n ← new Nodo                                                 ▷ O(1)
3:     n→izq ← NULL                                                          ▷ O(1)
4:     n→der ← NULL                                                          ▷ O(1)
5:     n→elem ← p                                                            ▷ O(1)
6:     n→padre ← NULL                                                        ▷ O(1)
7:     if c.raiz = NULL then                                                 ▷ O(1)
8:         c.raiz ← &n                                                       ▷ O(1)
9:     else
10:        if c.padreUlt = NULL then                                         ▷ O(1)
11:            c.padreUlt ← c.raiz                                           ▷ O(1)
12:            n→padre ← c.raiz                                              ▷ O(1)
13:            c.raiz→izq ← &n                                               ▷ O(1)
14:        else
15:            if c.padreUlt→der = NULL then                                 ▷ O(1)
16:                n→padre ← c.padreUlt                                      ▷ O(1)
17:                c.padreUlt→der ← &n                                       ▷ O(1)
18:            else
19:                int alt ← log2(c.cant)                                    ▷ O(1)
20:                if c.cant = 2^alt then                                    ▷ O(1)
21:                    var puntero(nodo) actual ← c.raiz                     ▷ O(1)
22:                    while actual→izq != NULL do                           ▷ O(log2(k))
23:                        actual ← actual→izq                               ▷ O(1)
24:                    end while
25:                    c.padreUlt ← actual                                   ▷ O(1)
26:                    n→padre ← actual                                      ▷ O(1)
27:                    actual→izq ← &n                                       ▷ O(1)
28:                else
29:                    var puntero(nodo) abuelo ← c.padreUlt→padre           ▷ O(1)
30:                    actual ← c.padreUlt                                   ▷ O(1)
31:                    while abuelo != NULL  abuelo→der = actual do          ▷ O(log2(k))
32:                        actual ← abuelo                                   ▷ O(1)
33:                        abuelo ← abuelo→padre                             ▷ O(1)
34:                    end while
35:                    actual ← abuelo→der                                   ▷ O(1)
36:                    while actual→izq != NULL do                           ▷ O(log2(k))
37:                        actual ← actual→izq                               ▷ O(1)
38:                    end while
39:                    c.padreUlt ← actual                                   ▷ O(1)
40:                end if
41:            end if
42:        end if
43:    end if
44:    acomodar(n)                                                           ▷ O(log2(k))
45:    c.cant++                                                              ▷ O(1)
46: end function
```

```
1: function iVacio( ) ⟶ res : estr                                   ▷ O(1)
2:     res ← < NULL, NULL, 0 >                                       ▷ O(1)
3: end function
```

```
1: function iVacia?(in c: estr) ⟶ res : bool                        ▷ O(1)
2:     res ← c.raiz = NULL                                           ▷ O(1)
3: end function
```

```
1: function Acomodar(in/out n: Nodo)                                 ▷ O(log2(k))
2:     var puntero(nodo) pad ← n→padre                               ▷ O(1)
3:     while pad != NULL && mayorPrioridad(n→elem, pad→elem) do      ▷ O(log2(k))
4:         swap(pad,n)                                               ▷ O(1)
5:         pad ← n→padre                                             ▷ O(1)
6:     end while
7: end function
```

```
1: function mayorPrioridad(in p1: paquete, in p2: paquete) ⟶ res : bool   ▷ O(1)
2:     res ← p1.prioridad > p2.prioridad                                  ▷ O(1)
3: end function
```

```
1: function Swap(in papa: Nodo, in hijo: Nodo)                       ▷ O(1)
2:     if papa→padre != NULL then                                    ▷ O(1)
3:         if papa→padre→der = papa then                             ▷ O(1)
4:             papa→padre→der ← hijo                                 ▷ O(1)
5:         else
6:             papa→padre→izq ← hijo                                 ▷ O(1)
7:         end if
8:     end if
9:     hijo→padre ← papa→padre                                       ▷ O(1)
10:    papa→padre ← hijo                                             ▷ O(1)
11:    if hijo→izq != NULL then                                      ▷ O(1)
12:        hijo→izq→padre ← papa                                     ▷ O(1)
13:    end if
14:    if hijo→der != NULL then                                      ▷ O(1)
15:        hijo→der→padre ← papa                                     ▷ O(1)
16:    end if
17:    var puntero(nodo) aux
18:    if papa→der = hijo then                                       ▷ O(1)
19:        aux ← papa→izq                                            ▷ O(1)
20:        aux→padre ← hijo                                          ▷ O(1)
21:        papa→der ← hijo→der                                       ▷ O(1)
22:        papa→izq ← hijo→izq                                       ▷ O(1)
23:        hijo→der ← papa                                           ▷ O(1)
24:        hijo→izq ← aux                                            ▷ O(1)
25:    else
26:        aux ← papa→der                                            ▷ O(1)
27:        aux→padre ← hijo                                          ▷ O(1)
28:        papa→der ← hijo→der                                       ▷ O(1)
29:        papa→izq ← hijo→izq                                       ▷ O(1)
30:        hijo→izq ← papa                                           ▷ O(1)
31:        hijo→der ← aux                                            ▷ O(1)
32:    end if
33: end function
```

```
1: function iDesencolar(in/out c: estr) ⟶ res : paquete                    ▷ O(log2(k))
2:     res ← c.raiz→elem                                                    ▷ O(1)
3:     var puntero(nodo) ult                                                ▷ O(1)
4:     if c.padreUlt→der != NULL then                                       ▷ O(1)
5:         ult ← c.padreUlt→der                                             ▷ O(1)
6:         c.padreUlt→der ← NULL                                            ▷ O(1)
7:     else
8:         ult ← c.padreUlt→izq                                             ▷ O(1)
9:         c.padreUlt→izq ← NULL                                            ▷ O(1)
10:    end if
11:    ult→padre ← NULL                                                     ▷ O(1)
12:    ult→izq ← c.raiz→izq                                                 ▷ O(1)
13:    ult→der ← c.raiz→der                                                 ▷ O(1)
14:    ult→der→padre ← ult                                                  ▷ O(1)
15:    ult→izq→padre ← ult                                                  ▷ O(1)
16:    c.raiz ← ult                                                         ▷ O(1)
17:    bajar(ult,c)                                                         ▷ O(log2(k))
18:    c.cant - -                                                           ▷ O(1)
19: end function
```

```
1: function Bajar(in/out c: estr)                                          ▷ O(log2(k))
2:     while n→izq != NULL && mayorPrioridad(n→elem, n→izq→elem) ∨n→der != NULL &&
   mayorPrioridad(n→elem, n→der→elem) do                                   ▷ O(log2(k))
3:         if n→der != NULL && mayorPrioridad(n→der→elem, n→izq→elem) then  ▷ O(1)
4:             if n = c.raiz then                                           ▷ O(1)
5:                 c.raiz ← n→der                                           ▷ O(1)
6:             end if
7:             swap(n, n→der)                                               ▷ O(1)
8:         else
9:             if n = c.raiz then                                           ▷ O(1)
10:                c.raiz ← n→izq                                           ▷ O(1)
11:            end if
12:            swap(n, n→izq)                                               ▷ O(1)
13:        end if
14:    end while
15: end function
```

7 Diseño del Tipo Conjunto Lineal Extendido(α)

7.1 Aspectos de la interfaz

7.1.1 Interfaz

Se extiende la interfaz del conjunto lineal dada en el apunte de módulos básicos.

Operaciones básicas de conjunto

UNIR(**in/out** c_1: $conj(\alpha)$, **in** c_2: $conj(\alpha)$)
Pre $\equiv$ { $c_1 =_{obs} c_0$}
Post $\equiv$ { $c_1 =_{obs} c_0 \cup c_2$}
Complejidad: $\mathcal{O}(n)$ *donde n es el cardinal del conjunto.*
Descripción: Devuelve la unión entre 2 conjuntos.

DAMEUNO(**in** c_1: $conj(\alpha)$) $\longrightarrow res : \alpha$
Pre $\equiv$ { $\neg$vacio(c)}
Post $\equiv$ { $res =_{obs} dameUno(c)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un elemento del conjunto c.

SINUNO(**in/out** c: $conj(\alpha)$)
Pre $\equiv$ { $\neg$vacio(c) $\wedge$ c $=_{obs}$ c_0 }
Post $\equiv$ { $res =_{obs} sinUno(c_0)$ }
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el conjunto c con un elemento menos.

7.1.2 Algoritmos

```
1: function iUNIR(in/out c1: conj(α), in c2: conj(α))                ▷ O(n)
2:     var itConj(α) it ← crearIt(c2)                                ▷ O(1)
3:     while (haySiguiente(it)) do                                   ▷ O(n)
4:         if (¬ pertenece?(c1, siguiente(it))) then                 ▷ O(1)
5:             agregar(c1,siguiente(it))                             ▷ O(1)
6:             avanzar(it)                                           ▷ O(1)
7:         end if
8:     end while
9: end function
```

```
1: function iDAMEUNO(in c: conj(α))⟶res :α                           ▷ O(1)
2:     res ← siguiente(crearIt(c))                                   ▷ O(1)
3: end function
```

```
1: function iSINUNO(in/out c: conj(α))                               ▷ O(1)
2:     eliminarSiguiente(crearIt(c1))                                ▷ O(1)
3: end function
```