

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	3
2. Módulo DCNet	9
3. Módulo Cola de Prioridad <i>HEAP</i> (α)	15
4. Módulo Diccionario <i>TRIE</i> (α)	20
5. Módulo Diccionario <i>AVL</i> (κ, σ)	21

1. Módulo Red

Interfaz

usa: $\text{CONJ}(\alpha)$, $\text{ITCONJ}(\alpha)$, $\text{LISTA}(\alpha)$, $\text{ITLISTA}(\alpha)$.

se explica con: RED.

géneros: red.

Operaciones de Red

COMPUTADORAS(**in** r : red) $\rightarrow res$: conj(hostname)

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

Post $\equiv \{res =_{obs} \text{dameHostnames}(\text{computadoras}(r))\}$

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 $\equiv \{c1, c2 \in \text{dameHostnames}(\text{computadoras}(r))\}$

Post $\equiv \{res =_{obs} \text{conectadas?}(r, \text{dameCompu}(c1), \text{dameCompu}(c2))\}$

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 $\equiv \{\text{conectadas?}(r, \text{dameCompu}(c1), \text{dameCompu}(c2))\}$

Post $\equiv \{res =_{obs} \text{interfazUsada}(r, \text{dameCompu}(c1), \text{dameCompu}(c2))\}$

Descripción: devuelve la interfaz por la cual estan conectadas $c1$ y $c2$.

INICIARRED() $\rightarrow res$: red

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

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

Descripción: crear una nueva Red.

AGREGARCOMPU(**in/out** r : red, **in** $c1$: compu)

Pre $\equiv \{r = r_0 \wedge (\forall c : \text{compu}) c \in \text{computadoras}(r_0) \Rightarrow \text{ip}(c) \neq c1\}$

Post $\equiv \{r =_{obs} \text{agregarComputadora}(r_0, c1)\}$

Descripción: agregar una computadora a la Red.

CONECTAR(**in** r : red, **in** $c1$: hostname, **in** $i1$: interfaz **in** $c2$: hostname, **in** $i2$: interfaz)

Pre $\equiv \{r = r_0 \wedge c1, c2 \in \text{dameHostnames}(\text{computadoras}(r)) \wedge c1 \neq c2 \wedge$

$\neg \text{conectadas?}(r, \text{dameCompu}(c1), \text{dameCompu}(c2)) \wedge \neg \text{usaInterfaz?}(r, \text{dameCompu}(c1), i1) \wedge$

$\neg \text{usaInterfaz?}(r, \text{dameCompu}(c2), i2)\}$

Post $\equiv \{r =_{obs} (\text{conectar}(r_0, \text{dameCompu}(c1), i1, \text{dameCompu}(c2), i2))\}$

Descripción: conectar dos computadoras de la red.

VECINOS(**in** r : red, **in** c : hostname) $\rightarrow res$: conj(hostname)

Pre $\equiv \{c \in \text{dameHostnames}(\text{computadoras}(r))\}$

Post $\equiv \{res =_{obs} \text{dameHostnames}(\text{vecinos}(r, \text{dameCompu}(c)))\}$

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 $\equiv \{c \in \text{dameHostnames}(\text{computadoras}(r))\}$

Post $\equiv \{res =_{obs} \text{usaInterfaz?}(r, \text{dameCompu}(c), i)\}$

Descripción: indica si la interfaz está siendo utilizada.

CAMINOSMINIMOS(**in** r : red, **in** $c1$: hostname, **in** $c2$: hostname) $\rightarrow res$: conj(lista(hostname))

Pre $\equiv \{c1, c2 \in \text{dameHostnames}(\text{computadoras}(r))\}$

Post $\equiv \{res =_{obs} \text{dameCaminosdeHostnames}(\text{caminosMinimos}(r, \text{dameCompu}(c_1), \text{dameCompu}(c_2)))\}$

Descripción: devuelve los conjuntos de caminos minimos entre las computadoras ingresadas.

Aliasing: res se devuelve por copia.

HAYCAMINO?(**in** $r : \text{red}$, **in** $c_1 : \text{hostname}$, **in** $c_2 : \text{hostname}$) $\rightarrow res : \text{bool}$

Pre $\equiv \{c_1, c_2 \in \text{dameHostnames}(\text{computadoras}(r))\}$

Post $\equiv \{res =_{obs} \text{hayCamino?}(r, \text{dameCompu}(c_1), \text{dameCompu}(c_2))\}$

Descripción: indica si las computadoras son alcanzables mediante algún camino.

• == •(**in** $r_1 : \text{red}$, **in** $r_2 : \text{red}$) $\rightarrow res : \text{bool}$

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

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

TAD RED EXTENDIDA

extiende RED

otras operaciones

(no exportadas)

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

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

dameHostnames(cc) \equiv **if** vacio?(cc) **then**
 \emptyset

else

Ag(ip(dameUno(cc)), dameHostnames(sinUno(cc)))

fi

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

auxDameCompu(r, s, cc) \equiv **if** ip(dameUno(cc)) = s **then**
dameUno(cc)

else

auxDameCompu($r, s, \text{sinUno}(cc)$)

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: lista(lista(hostname))))

Rep : *estr* \longrightarrow bool

Rep(*e*) \equiv true \iff ...

Abs : *estr* *r* \longrightarrow red {Rep(*r*)}

Abs(*r*) \equiv ...

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 IGUALDAD(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
                                        end while
                                        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
                            Avanzar(itAlc1) ▷ O(1)
                        end while
                    end if
                end while
            end if
        end while
        Avanzar(itRed1) ▷ O(1)
    end while
    end if
end function

```

2. Módulo DCNet

Interfaz

usa: RED, CONJ(α), ITCONJ(α), LISTA(α), ITLISTA(α), DICT_{TRIE}(κ , σ), DICT_{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 $\equiv \{\text{true}\}$

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

Descripción: devuelve la red asociada.

Aliasing: res no es modificable.

CAMINORECORRIDO(**in** d : dcnet, **in** p : IDpaquete) $\rightarrow res$: lista(hostname)

Pre $\equiv \{\text{IDpaqueteEnTransito?}(d, p)\}$

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

Descripción: devuelve los paquetes en la cola de la computadora.

Aliasing: res no es modificable.

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

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

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 $\equiv \{d = d_0 \wedge \neg(\exists p': \text{paquete}) (\text{paqueteEnTransito?}(d, p') \wedge \text{id}(p') = \text{id}(p)) \wedge$

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

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

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

Descripción: indica si el paquete esta en alguna de las colas dado el ID.

LAQUEMASENVIO(in $d: \text{dcnet}$) $\rightarrow res: \text{hostname}$

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 $d_1: \text{dcnet}$, in $d_2: \text{dcnet}$) $\rightarrow res: \text{bool}$

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

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

dameHostnames(cc) \equiv **if** vacio?(cc) **then**
 \emptyset
else
 Ag(ip(dameUno(cc)), dameHostnames(sinUno(cc)))
fi
 dameCompu(d, s) \equiv auxDameCompu($s, \text{computadoras}(\text{red}((d)))$)
 auxDameCompu(s, cc) \equiv **if** ip(dameUno(cc)) = s **then**
 dameUno(cc)
else
 auxDameCompu($s, \text{sinUno}(cc)$)
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
 todos: conj(X)
 porHostname: dicc_{TRIE} (hostname, itConj(X))
 conMasEnvios: itConj(X)
 proxDest: arreglo_dimensionable de arreglo_dims. de
 lista(compu))

donde X es tupla(*compu*: hostname
 indice: nat
 paquetes: conj(paquete)
 cola: conj_{HEAP}(itConj(paquete))
 paqPorID: dicc_{AVL} (IDpaquete, itConj(paquete))
 cantEnvios: nat)

Rep : *estr_dcnet* \longrightarrow bool

Rep(*e*) \equiv true \iff

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

Abs(*d*) \equiv ...

Algoritmos

Algorithm 12 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 13 Implementación de LaQueMasEnvío

```

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

```

Algorithm 14 Implementación de AvanzarSegundo

```

function IAVANZARSEGUNDO(inout d: estr_dcnet)
  arreglo  $\leftarrow$  crearArreglo[#Claves(d.computadoras)] de tupla(usedo: bool, paquete: paquete, destino:
  string), donde paquete es tupla(IDpaquete: nat, prioridadL nat, origen: string, destino: string)
                                      $\triangleright$   $O(n)$  para calcular cantidad de claves,  $O(1)$  para crearlo
  for (int i=0, < #Claves(d.computadoras), i++) do                                      $\triangleright$  el ciclo se hará n veces
    arreglo[i].usedo = false                                                          $\triangleright$   $O(1)$ 
  end for

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

```

function IGUALDAD(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 $_{HEAP}(\alpha)$

Interfaz

usa: TUPLA, NAT, BOOL, α .

se explica con: COLA DE PRIORIDAD ALTERNATIVA.

géneros: heap.

Operaciones de Cola de Prioridad $_{HEAP}$

VACIA $(\rightarrow res : \text{heap})$

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

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

Complejidad: $O(1)$

Descripción: Crea una cola vacía.

VACIA? $(\text{in } estr : \text{heap}) \rightarrow res : \text{bool}$

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

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

Complejidad: $O(1)$

Descripción: Indica si la cola esta vacía.

ENCOLAR $(\text{in/out } estr : \text{heap}, \text{in } prio : \text{nat}, \text{in } valor : \alpha) \rightarrow res : \text{bool}$

Pre $\equiv \{estr = estr_0\}$

Post $\equiv \{res \wedge estr =_{obs} \text{encolar}(estr_0)\}$

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

Descripción: Crea un nuevo elemento con los parametros dados y lo agrega a la cola.

DESENCOLAR $(\text{in/out } estr : \text{heap}) \rightarrow res : \alpha$

Pre $\equiv \{estr = estr_0 \wedge \neg \text{Vacía?}(estr)\}$

Post $\equiv \{estr =_{obs} \text{desencolar}(estr_0) \wedge res =_{obs} \text{proximo}(estr_0)\}$

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 $\text{colaPrio}(\alpha)$

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

igualdad observacional

$$\left((\forall c_1, c_2 : \text{colaPrio}(\alpha)) (c_1 =_{obs} c_2 \iff (\forall n : \text{nat}) \text{conjPorPrio}(c_1, n) =_{obs} \text{conjPorPrio}(c_2, n)) \right)$$

usa $\text{BOOL}, \text{NAT}, \text{CONJUNTO}(\alpha)$

observadores básicos

prioridades : $\text{colaPrio}(\alpha) \longrightarrow \text{conj}(\text{nat})$

conjPorPrio : $\text{colaPrio}(\alpha) \times \text{nat} \longrightarrow \text{conj}(\alpha)$

próximo : $\text{colaPrio}(\alpha) \longrightarrow \alpha$ $\{-\emptyset?(\text{prioridades}(c))\}$

generadores

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

$\text{encolar} : \text{colaPrio}(\alpha) \times \text{nat} \times \alpha \longrightarrow \text{colaPrio}(\alpha)$
 $\text{desencolar} : \text{colaPrio}(\alpha) \ c \longrightarrow \text{colaPrio}(\alpha) \quad \{-\emptyset?(prioridades(c))\}$

otras operaciones

$\text{próximaPrio} : \text{colaPrio}(\alpha) \longrightarrow \text{nat}$
 $\text{maxConjunto} : \text{conjunto}(\text{nat}) \ C \longrightarrow \text{nat}$

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

$\text{próximaPrio}(c) \equiv \text{maxConjunto}(\text{prioridades}(c))$

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

$\text{prioridades}(vacía) \equiv \emptyset$

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

$\text{prioridades}(\text{desencolar}(c)) \equiv \text{if } \#conjPorPrio(c, \text{proximaPrio}(c)) = 1 \text{ then } \text{prioridades}(c) -$
 $\{\text{proximaPrio}(c)\} \text{ else } \text{prioridades}(c)$

$\text{conjPorPrio}(vacía, p) \equiv \emptyset$

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

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

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

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

Fin TAD

Representación

heap se representa con *estr*

donde *estr* es $\text{tupla}(\text{size: nat}$
 $\quad \text{primero: nodo}(\alpha)$
 $\quad)$
 donde $\text{nodo}(\alpha)$ es $\text{tupla}(\text{padre: puntero}(\text{nodo}(\alpha))$
 $\quad \text{izq: puntero}(\text{nodo}(\alpha))$
 $\quad \text{der: puntero}(\text{nodo}(\alpha))$
 $\quad \text{prio: nat}$
 $\quad \text{valro: } \alpha$
 $\quad)$

$\text{Rep: } \text{estr} \rightarrow \text{bool}$
 $\text{Rep}(e) \equiv \text{true} \iff \text{size} = \# \text{arbol}(\text{estr.primero}) \wedge_L$
 $((\text{estr.primero}).\text{padre} = \text{null} \wedge$
 $(\forall e : \text{estr})(e \in \text{arbol}(\text{estr.primero}) \wedge e \neq (\text{estr.primero}) \Rightarrow (e.\text{padre} \neq \text{null} \wedge_L (((e.\text{padre}).\text{izq} =$
 $e \vee (e.\text{padre}).\text{der} = e) \wedge \neg(((e.\text{padre}).\text{izq} = e \wedge (e.\text{padre}).\text{der} = e)))) \wedge$
 $(\forall e : \text{estr})(e \in \text{arbol}(\text{estr.primero}) \Rightarrow ((\text{estr.izq} \neq \text{null} \Rightarrow \text{estr.prio} \geq (\text{estr.izq}).\text{prio}) \wedge (\text{estr.der} \neq$
 $\text{null} \Rightarrow \text{estr.prio} \geq (\text{estr.der}).\text{prio}))) \wedge$
 $(\forall e : \text{estr})(e \in \text{arbol}(\text{estr.primero}) \Rightarrow \text{caminoHastaRaiz}(e, \text{arbol}(\text{estr.primero})) \leq \lfloor \log_2(\text{size}) \rfloor + 1))$
 $\text{Abs: } \text{estr } d \rightarrow \text{colaPrio}(\alpha) \{ \text{Rep}(d) \}$
 $\text{Abs}(\text{Vacía}()) \equiv \text{Vacía}$
 $(\forall e : \text{estr}, n : \text{nat}, \text{valor} : \alpha)(\text{Abs}(\text{encolar}(e, n, \text{valor}))) \equiv \text{encolar}(\text{Abs}(e), n, \text{valor})$
 $(\forall e : \text{estr})(\text{Abs}(\text{desencolar}(e))) \equiv \text{desencolar}(\text{Abs}(e))$

Especificación de las operaciones auxiliares utilizadas para Rep y Abs

$\text{arbol} : \text{nodo}(\alpha) \rightarrow \text{conj}(\text{nodo}(\alpha))$
 $\text{caminoHastaRaiz} : \text{nodo}(\alpha) \rightarrow \text{nat}$
 $\text{arbol}(n) \equiv \text{if } n.\text{izq} \neq \text{null} \wedge n.\text{der} \neq \text{null} \text{ then } \text{Ag}(n.\text{valor}, \text{arbol}(n.\text{izq}) \cup \text{arbol}(n.\text{der})) \text{ else if } n.\text{izq} \neq$
 $\text{null} \text{ then } \text{Ag}(n.\text{valor}, \text{arbol}(n.\text{izq})) \text{ else if } n.\text{der} \neq \text{null} \text{ then } \text{Ag}(n.\text{valor}, \text{arbol}(n.\text{der})) \text{ else }$
 $\text{Ag}(n.\text{valor}, \emptyset)$
 $\text{caminoHastaRaiz}(n) \equiv \text{if } n.\text{padre} = \text{null} \text{ then } 0 \text{ else } \text{caminoHastaRaiz}(n.\text{padre}) + 1$

Algoritmos

Algorithm 16 Implementación de Vacía

```
function iVACIA → res: heap
  res ← <0, null>
end function
```

▷ O(1)

Algorithm 17 Implementación de Vacía?

```
function iVACIA?(in estr: heap) → res: bool
  res ← (estr.primero == null)
end function
```

▷ O(1)

Algorithm 18 Implementación de Encolar

```

function IENCOLAR(in/out estr : heap, in prio : nat, in valor :  $\alpha$ )  $\rightarrow$  res : bool
    res  $\leftarrow$  true  $\triangleright$  O(1)
    if estr.size == 0 then  $\triangleright$  O(1)
        estr.primer  $\leftarrow$  <null, null, null, prio, valor>  $\triangleright$  O(1)
    else
        size ++  $\triangleright$  O(1)
        x  $\leftarrow$  valor  $\triangleright$  O(1)
        y  $\leftarrow$  <>  $\triangleright$  O(1)
        while x  $\neq$  0 do  $\triangleright$  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) • y  $\triangleright$  O(1)
            x  $\leftarrow$  x / 2  $\triangleright$  O(1)
        end while
        y  $\leftarrow$  com(y)  $\triangleright$  O(log(n))
        z  $\leftarrow$  estr.primer  $\triangleright$  O(1)
        y  $\leftarrow$  fin(y)  $\triangleright$  O(1)
        while long(y) > 1 do  $\triangleright$  El ciclo se ejecuta O(log(n)) veces
            z  $\leftarrow$  if prim(y) == 0 then z.izq else z.der  $\triangleright$  O(1)
            y  $\leftarrow$  fin(y)  $\triangleright$  O(1)
        end while
        w  $\leftarrow$  <null, null, null, prio, valor>  $\triangleright$  O(1)
        w.padre  $\leftarrow$  z  $\triangleright$  O(1)
        if prim(y) == 0 then  $\triangleright$  O(1)
            z.izq  $\leftarrow$  w  $\triangleright$  O(1)
        else
            z.der  $\leftarrow$  w  $\triangleright$  O(1)
        end if
        while w  $\neq$  estr.primer  $\wedge_L$  w.prio > (w.padre).prio do  $\triangleright$  La cantidad de veces que se ejecuta el
            ciclo es a lo sumo la altura del heap, que es O(log(n))
            aux  $\leftarrow$  w.valor  $\triangleright$  O(1)
            w.valor  $\leftarrow$  (w.padre).valor  $\triangleright$  O(1)
            (w.padre).valor  $\leftarrow$  aux  $\triangleright$  O(1)
            w  $\leftarrow$  w.padre  $\triangleright$  O(1)
        end while
    end if
end function

```

Algorithm 19 Implementación de Desencolar

```

function IDESENCOLAR(in/out estr : heap)  $\rightarrow$  res :  $\alpha$ 
    res  $\leftarrow$  estr.primer                                 $\triangleright O(1)$ 
    x  $\leftarrow$  valor                                        $\triangleright O(1)$ 
    y  $\leftarrow$   $\langle \rangle$                                           $\triangleright O(1)$ 
    while x  $\neq$  0 do  $\triangleright$  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) • y                                 $\triangleright O(1)$ 
        x  $\leftarrow$  x / 2                                      $\triangleright O(1)$ 
    end while
    y  $\leftarrow$  com(y)                                          $\triangleright O(\log(n))$ 
    z  $\leftarrow$  estr.primer                                      $\triangleright O(1)$ 
    y  $\leftarrow$  fin(y)                                          $\triangleright O(1)$ 
    while long(y) > 1 do  $\triangleright$  El ciclo se ejecuta  $O(\log(n))$  veces
        z  $\leftarrow$  if prim(y) == 0 then z.izq else z.der     $\triangleright O(1)$ 
        y  $\leftarrow$  fin(y)                                      $\triangleright O(1)$ 
    end while
    w  $\leftarrow$   $\langle \text{null}, \text{null}, \text{null}, \text{prio}, \text{valor} \rangle$      $\triangleright O(1)$ 
    w.padre  $\leftarrow$  z                                        $\triangleright O(1)$ 
    if prim(y) == 0 then                                      $\triangleright O(1)$ 
        z.izq  $\leftarrow$  w                                      $\triangleright O(1)$ 
    else
        z.der  $\leftarrow$  w                                      $\triangleright O(1)$ 
    end if
    (estr.primer).valor  $\leftarrow$  z.valor                      $\triangleright O(1)$ 
    borrar(z)                                               $\triangleright O(1)$ 
    size  $\leftarrow$   $\text{--}$                                           $\triangleright O(1)$ 
    while (z.izq  $\neq$  null  $\vee$  z.der  $\neq$  null)  $\wedge_L$  z.valor < maxValor(z.izq, z.der) do  $\triangleright$  La cantidad de
    veces que se ejecuta el ciclo es a lo sumo la altura del heap, que es  $O(\log(n))$ 
         $\triangleright$  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  $\triangleright O(1)$ 
        aux  $\leftarrow$  z.valor                                 $\triangleright O(1)$ 
        z.valor  $\leftarrow$  (z.izq).valor                      $\triangleright O(1)$ 
        (z.izq).valor  $\leftarrow$  aux                            $\triangleright O(1)$ 
        z  $\leftarrow$  z.izq                                     $\triangleright O(1)$ 
    else
        aux  $\leftarrow$  z.valor                                 $\triangleright O(1)$ 
        z.valor  $\leftarrow$  (z.der).valor                      $\triangleright O(1)$ 
        (z.der).valor  $\leftarrow$  aux                            $\triangleright O(1)$ 
        z  $\leftarrow$  z.der                                     $\triangleright O(1)$ 
    end if
end function

```

4. Módulo Diccionario $TRIE$ (α)

Interfaz

usa: .

se explica con: COLA DE PRIORIDAD ALTERNATIVA.

Operaciones

Operaciones

DEFINIDA(**in** $d: \text{dicc}_{TRIE}(\text{clave}, \text{significado})$, **in** $c: \text{clave}$) $\rightarrow res: \text{Bool}$

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

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

Complejidad: $O(L)$

OBTENER(**in** $d: \text{dicc}_{TRIE}(\text{clave}, \text{significado})$, **in** $c: \text{clave}$) $\rightarrow res: \text{significado}$

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

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

Complejidad: $O(L)$

VACIO() $\rightarrow res: \text{dicc}_{TRIE}(\text{clave}, \text{significado})$

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

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

Complejidad: $O(1)$

DEFINIR(**in/out** $d: \text{dicc}_{TRIE}(\text{clave}, \text{significado})$, **in** $c: \text{clave}$, **in** $s: \text{significado}$) $\rightarrow res: \text{Bool}$

Pre $\equiv \{\neg \text{def?}(c,d) \wedge d=d_0\}$

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

Complejidad: $O(L)$

BORRAR(**in/out** $d: \text{dicc}_{TRIE}(\text{clave}, \text{significado})$, **in** $c: \text{clave}$) $\rightarrow res: \text{Bool}$

Pre $\equiv \{d=d_0\}$

Post $\equiv \{d=\text{borrar}(c,d_0)\}$

Complejidad: $O(L)$

Descripción: Devuelve TRUE si se pudo borrar la clave, o FALSE si no la encontró

CLAVES(**in** $d: \text{dicc}_{TRIE}(\text{clave}, \text{significado})$) $\rightarrow res: \text{conj}(\text{clave})$

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

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

Complejidad: $O()$

5. Módulo Diccionario $_{AVL} (\kappa, \sigma)$

Interfaz

usa: .

se explica con: DICcionario(CLAve,SIGNIFICADO).

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

Operaciones

DEFINIDO?(in d : dicc $_{AA}$ (clave, significado), in c : clave) $\rightarrow res$: Bool

Pre $\equiv \{\}$

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

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

SIGNIFICADO(in d : dicc $_{AA}$ (clave,significado), in c : clave) $\rightarrow res$: significado

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

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

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

VACIO() $\rightarrow res$: dicc $_{AA}$ (clave,significado)

Pre $\equiv \{true\}$

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

Complejidad: $O(1)$

DEFINIR(in/out d : dicc $_{AA}$ (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 : dicc $_{AA}$ (clave,significado), in c : clave)

Pre $\equiv \{def?(c,d) \wedge d=d_0\}$

Post $\equiv \{d=borrar(c,d_0)\}$

Complejidad: $O()$

Representación

dicc $_{AA}$ se representa con estr $_{AA}$

donde:

estr $_{AA}$ es dicc $_{AA}$ (id, it)

donde datos es tupla(id : nat

izquierdo: puntero(nodo(id , it))

derecho: puntero(nodo(id , it))

valor: itConj(α)

altura: nat)

Rep : estr \rightarrow bool

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

Abs : estr $r \rightarrow$ dicc $_{AA}$ (id, it)

{Rep(r)}

Abs(r) $\equiv \dots$

Algoritmos

Algorithm 20 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
      else nodoActual ← nodoActual.derecho
    end if
  end if
end while
end function
```

Algorithm 21 Implementación de Significado

```
function ISIGNIFICADO(in d: estr_AA , in c: clave)→ res:  $\alpha$ 
  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
      else nodoActual ← nodoActual.derecho
    end if
  end if
end while
end function
```

Algorithm 22 Implementación de Vacio

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

Algorithm 23 Implementación de Definir

```

function IDEFINIR(in d: estr_AA , in c: nat, in s:  $\alpha$ )
  nodoActual  $\leftarrow$  d
  yaDefini  $\leftarrow$  FALSE
  while  $\neg$ yaDefini do
    if  $c < \text{nodoActual.id}$  then
      if  $\text{nodoActual.izquierdo} == \text{NULO}$  then
        nuevoNodo  $\leftarrow$  Vacio()
        nuevoNodo.valor  $\leftarrow$  Copiar(s)
        nuevoNodo.id  $\leftarrow$  c
        nodoActual.izquierdo  $\leftarrow$  nuevoNodo
        yaDefini gets TRUE
      else
        nodoActual  $\leftarrow$  nodoActual.izquierdo
      end if
    else
      if  $\text{nodoActual.derecho} == \text{NULO}$  then
        nuevoNodo  $\leftarrow$  Vacio()
        nuevoNodo.valor  $\leftarrow$  Copiar(s)
        nuevoNodo.id  $\leftarrow$  c
        nodoActual.derecho  $\leftarrow$  nuevoNodo
        yaDefini gets TRUE
      else
        nodoActual  $\leftarrow$  nodoActual.derecho
      end if
    end if
  end while
  nodoActual  $\leftarrow$  Torsion(nodoActual)
  nodoActual  $\leftarrow$  Division(nodoActual)
end function

```

 \triangleright Dicc_AA \triangleright Dicc_AA