Algoritmos y Estructuras de Datos II

Segundo Cuatrimestre de 2009

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

Trabajo Práctico 2 (Recuperatorio)

Diseño de Tipos Abstractos de Datos

Enrutador de tráfico IP

Grupo 7

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	marianobianchi08@gmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar
Hernandez, Fernando	—	matematica527@yahoo.es

Aclaraciones

Extension del Secu(α)

El Tad SecuOrdSinRep(α) es una extension del Tad Secu(α) en la cual se agregan algunas operaciones y se reemplaza el generador $\bullet$ por agregarOdenado. En consecuencia, las operaciones hechas sobre el operador puntito pierden validez.

Ordenes de los elementos α

Durante este trabajo práctico se dió por sentado que se tenían los siguientes órdenes totales:

- En la estructura del enrutador, el campo *reglas* es de tipo secuOrdSinRep. Las tuplas que esta contiene, se ordenan por versión, es decir, se ordenan según el orden dado por el primer campo de cada tupla.
- En las secuencias de eventos, se da por sentado que el orden de los mismos está dado por el timestamp, ordenándose de forma creciente.

Conjunto de interfaces

Como se indica en el enunciado, decidimos reemplazar al conjunto de interfaces por una cantidad máxima de interfaces, las cuales van de 0 a cantInterfaces-1.

Aliasing Router

En la interfaz del router, los siguientes parámetros sean de tipo in, inout o devueltos como resultado, son pasados por referencia:

- Router
- ReglaDir
- Evento
- DirIp
- Conjunto(α)
- SecuOrdSinRep(α)

El resto de los parámetros, son tipos básicos, por lo cual son pasados por copia.

En el caso de los elementos devueltos como resultado, si no son de alguno de los tipos compuestos aclarados anteriormente, siempre se devuelven por copia.

Aliasing SecuOrdSinRep

En la interfaz de la secuOrdSinRep, cuando se recibe un parámetro de tipo α, si es un tipo básico, se pasa por copia, mientras que si es un tipo complejo, se pasa por referencia.

El resto de los parámetros, se pueden considerar parámetros básicos, por lo cual se pasan por copia.

En el caso de los elementos devueltos como resultado, siempre se devuelven por copia si son de tipos básicos, en caso de devolver una Secuencia, esta se devuelve por referencia.

Aliasing Conjunto

En la interfaz del conjunto, los parámetros recibidos como in o inout de tipo Conjunto, son pasados por referencia.

En el caso de los elementos paramétricos, si el parámetro es un tipo básico, se pasa por copia, mientras que si es complejo, se pasa por referencia.

En el caso de los elementos devueltos como resultado, siempre se devuelven por copia si son de tipos básicos, en caso de devolver un Conjunto, este se devuelve por referencia.

Aliasing Arbol

En la interfaz del arbolDeReglas, los siguientes parámetros sean de tipo in o inout, son pasados por referencia:

- arbolDeReglas
- ReglaDir
- DirIp

En el caso de los elementos devueltos como resultado, siempre se devuelven por copia.

Descripcion de funciones utilizadas

Las siguientes funciones fueron utilizadas para describir Ordenes de complejidad o funciones de representación o abstracción. Las mismas están incluidas dentro de los TAD correspondientes, pero no tienen ninguna función análoga dentro de los módulos presentados para este trabájo práctico. Además, la función altura solo se utiliza para la función de representación del Enrutador y rompería la igualdad observacional del TAD arbolDeReglas, por lo cual la misma se encuentra axiomatizada en este apartado.

$\text{posicionRelativa} : \text{secuOrdSinRep}(\alpha) \times \alpha \longrightarrow \text{nat}$

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

$\text{altura} : \text{ArbolDeReglas} \longrightarrow \text{nat}$

$\text{altura}(\text{nuevo}()) \equiv 0$

$\text{altura}(\text{agRegla}(a,r)) \equiv \max(\text{cantBits}(r),\text{altura}(r))$

Modulo Router

Interfaz

interfaz *Router*
se explica con la especificación de *Router*
usa interfaces BOOL, NAT, CONJUNTO(VERSIÓN), SECUENCIA(EVENTO), REGLADIR, RESPUESTADIR, EVENTO, VERSIÓN, INTERFAZ, DIRIP
genero *router*

operaciones

nuevo(in n: nat)→ res: router (O(n))
{true}
{$res =_{\text{obs}}$ nuevo(n)}

agVersión(inout r: router, in v: versión) (O(v))
{$(r =_{\text{obs}} r_0) \wedge (v \notin \text{versiones}(r))$}
{$r =_{\text{obs}}$ agVersión(r_0,v)}

agRegla(inout r: router, in u: reglaDir) (O(version(dirIp(u))))
{$(r =_{\text{obs}} r_0) \wedge (\text{versión}(\text{dirIp}(u)) \in \text{versiones}(r)) \wedge (\text{inter}(u)\ \text{interfaces}(r))$}
{$r =_{\text{obs}}$ agRegla(r_0,u)}

agEvento(inout r: router, in e :evento) (O(a)[1])
{$(r =_{\text{obs}} r_0) \wedge (\neg\exists$ i: interfaz)(i $<$ interfaces(r)) $\wedge_L$ existeTimestamp?(eventos(r,i),timestamp(e)))}
{$r =_{\text{obs}}$ agEvento(r_0,e)}

versiones(in r: router)⟶ res: conj(versión) (O(1))
{true}
{$res =_{\text{obs}}$ versiones(r)}

interfaces(in r: router)⟶ res: nat (O(1))
{true}
{$res =_{\text{obs}}$ interfaces(r)}

enrutar(in r: router, in d: dirIp)⟶ res: respDir (O(version(d)))
{true}
{$res =_{\text{obs}}$ enrutar(r,d)}

eventos(in r: router, in i: interfaz)⟶ res: secuOrdSinRep(eventos) (O(1))
{true}
{$res =_{\text{obs}}$ eventos(r,i)}

estaCaida?(in r: router, in i: interfaz)⟶ res: bool (O(1))
{$i <$ interfaces(r)}

[1] a = longitud(eventos(r,inter(e)) - posicionRelativa(eventos(r,inter(e)), e)

$\{res =_{\text{obs}} \text{estaCaida}(r,i)\}$

tiempoCaida(in r: router, in i: interfaz)$\longrightarrow$ res: nat (O(1))
$\{i < \text{interfaces}(r)\}$
$\{res =_{\text{obs}} \text{tiempocaida}(r,i)\}$

Representación

enrutador **se representa con** *estr*
donde *estr* **es** *tupla* $\langle$ *versiones: conj(versión)*
$\times$ *cantInterfaces: nat*
$\times$ *reglas: secuOrdSinRep(tupla⟨version: version, aR: abr⟩)*
$\times$ *status_inter: ad(tupla⟨tiempoCaida: nat, eventos: secuOrdSinRep(evento)⟩)* $\rangle$

Invariante de Representación

Rep : estr $\longrightarrow$ bool

$(\forall$ e:estr)Rep(e) = versionesDeReglasValidas(e.reglas, e.versiones)
$\wedge$ noFaltanVersionesEnReglas(e)
$\wedge_{\text{L}}$ interfacesDeArbolesValidas(e.reglas, e.cantInterfaces)
$\wedge$ alturaDeArbolesValida(e.reglas)
$\wedge$ longitudDeEventosValida(e.status_inter, e.cantInterfaces)
$\wedge_{\text{L}}$ interfazDeEventosCorrecta(e.status_inter)
$\wedge_{\text{L}}$ tiempoCaidaValido(e.status_inter)

funciones auxiliares:

versionesDeReglasValidas : secuOrdSinRep(tupla⟨version:version $\times$ aR:abr⟩) $\times$ conj(version) $\longrightarrow$ bool

versionesDeReglasValidas(s,c) $\equiv$ **if** vacia?(s) **then** *true*
else prim(s).version $\in$ c $\wedge$ versionesDeReglasValidas(fin(s),c)

noFaltanVersionesEnReglas : estr $\longrightarrow$ bool

noFaltanVersionesEnReglas(e) $\equiv$ $(\forall$ v: version)(v $\in$ e.versiones $\Rightarrow_{\text{L}}$estaVersionEnTupla(v,e.reglas))

interfacesDeArbolesValidas : secuOrdSinRep(tupla⟨version:version $\times$ aR:abr⟩) $\times$ nat $\longrightarrow$ bool

interfacesDeArbolesValidas(s,n) $\equiv$ $(\forall$ d: dirIp) (estaVersionEnTupla(version(d),s) $\Rightarrow_{\text{L}}$
interValidasAux(s,n,d))

estaVersionEnTupla : version $\times$ secuOrdSinRep(tupla⟨version:version $\times$ aR:abr⟩) $\longrightarrow$ bool

estaVersionEnTupla(v,s) $\equiv$ **if** vacia?(s) **then** *false*
else prim(s).version = v $\vee$ estaVersionEnTupla(v,fin(s))

interValidasAux : secuOrdSinRep(tupla⟨version:version × aR:abr⟩) s × nat × dirIp $d \longrightarrow$ bool
estaVersionEnTupla(version(d),s)

interValidasAux(s,n,d) $\equiv$ **if** prim(s).version = version(d)
then interfazDeSalida(prim(s).aR,d) < n
else interValidasAux(fin(s),n,d)

alturaDeArbolesValida : secuOrdSinRep(tupla⟨version:version × aR:abr⟩) $\longrightarrow$ bool

alturaDeArbolesValida(s) $\equiv$ **if** vacia?(s) **then** true
else 8 * prim(s).version $\geq$ altura(prim(s).aR)
$\wedge$ alturaDeArbolesValida(fin(s))

interfazDeEventosCorrecta : ad(tupla(nat × secuOrdSinRep(evento))) $\longrightarrow$ bool

interfazDeEventosCorrecta(a) $\equiv$ ($\forall$ i:interfaz) (i < tam(a) $\Rightarrow_{\text{L}}$ sonEventosDeLaInterfaz(a[i].eventos,i))

sonEventosDeLaInterfaz : secuOrdSinRep(evento) × interfaz $\longrightarrow$ bool

sonEventosDeLaInterfaz(s,i) $\equiv$ **if** vacia?(s) **then** *true*
else inter(prim(s)) = i $\wedge$ sonEventosDeLaInterfaz(fin(s),i)

tiempoCaidaValido : ad(tupla(nat × secu(evento))) $\longrightarrow$ bool

tiempoCaidaValido(a) $\equiv$ ($\forall$ i:interfaz) (i < tam(a) $\Rightarrow_{\text{L}}$ a[i].tiempoCaida = sumarCaidas(a[i].eventos)

sumarCaidas : secuOrdSinRep(evento) $\longrightarrow$ nat

sumarCaidas(s) $\equiv$ **if** vacia?(s) $\vee_{\text{L}}$ vacia?(fin(s)) **then** 0
else if caida?(ult(s))
then timestamp(ult(com(s))) - timestamp(ult(s)) + sumar-
Caidas(com(s))
else sumarCaidas(com(s))

longitudDeEventosValida : ad(tupla(nat × secuOrdSinRep(evento))) × nat $\longrightarrow$ bool

longitudDeEventosValida(a,n) $\equiv$ tam(a) = n

Función de Abstracción

Abs : estr e $\longrightarrow$ enrutador Rep(e)

($\forall$ e: estr) Abs(e) = r: router/
versiones(r) $=_{obs}$ e.versiones $\wedge$ interfaces(r) $=_{obs}$ e.cantInterfaces $\wedge_{L}$
($\forall$ i: interfaz)(i < interfaces(r) $\Rightarrow_{L}$eventos(r,i) $=_{obs}$ (e.status_inter[i]).eventos) $\wedge$
($\forall$ d: dirIp)(enrutar(r,d) $=_{obs}$ dameRtaDir(e,d))

funciones auxiliares

dameRtaDir : estr e $\times$ dirIp $\longrightarrow$ respuestaDir Rep(e)

dameRtaDir(e,d) $\equiv$ **if** version(d) $\notin$ e.versiones **then** DireccionNoSoportada()
else if $\neg$tieneRegla(arbolDeLaVersion(version(d),e.reglas),d)
then interfazDeSalidaNoEncontrada()
else if estaCaida?(prim((e.status_inter[queInterfaz(e,d)]).eventos))
then interfazDeSalidaCaida(queInterfaz(e,d))
else SalidaPorInterfaz(queInterfaz(e,d))

queInterfaz : estr e $\times$ dirIp $\longrightarrow$ interfaz Rep(e)

queInterfaz(e,d) $\equiv$ interfazDeSalida(arbolDeLaVersion(version(d),e.reglas),d)

arbolDeLaVersion : version v $\times$ secu(tupla(version $\times$ abr)) s $\longrightarrow$ abr

arbolDeLaVersion(v,s) $\equiv$ **if** vacia?(s) then nuevo()
else if prim(s).version = v then prim(s).aR
else arbolDeLaVersion(v,fin(s))

Algoritmos

*i*nuevo(in n: nat) $\longrightarrow$ res: estr

```
1 Complejidad: O(n)
2 var inter: arreglo_dimensionable de tupla⟨nat, secuOrdSinRep(evento)⟩
3 inter ← CrearArreglo(n);                                          // O(n)
4 for (i: nat ← 0; i < n;i++) ;                                     // O(n)
5 do
6 |  inter[i] ← ⟨0, <>⟩;                                           // O(1)
7 end
8 res ← ⟨∅, n, <>, inter⟩;                                          // O(1)
```

*i*agVersion(inout e: estr, in v: version)

```
1 Complejidad: O(v)
2 agregar(v, e.versiones);                                          // O(v)[a]
3 insertarOrdenadoDesdeAdelante(⟨v, <>⟩, e.reglas);                 // O(v)[b]
```

[a] La complejidad es del orden de posicionRelativa(e.versiones, v) como lo indica la interfaz de Conjunto, pero en el peor caso, posicionRelativa(e.versiones, v) = v

[b] La complejidad es del orden de posicionRelativa(e.versiones, v) como lo indica la interfaz de SecuOrdSinRep, pero en el peor caso, posicionRelativa(e.reglas, ⟨v, <>⟩) = v

*i*agRegla(inout e: estr, in u: reglaDir)

```
1 Complejidad: O(version(dirIp(u)))
2 var it: itOrd(secuOrdSinRep(tupla ⟨version, abr⟩))
3 var v: version
4 it ← crearIt(e.reglas) ;                                          // O(1)
5 v ← u.version ;                                                   // O(1)
6 while (actualAdelante(it).version ≠ v) ;                          // O(version(dirIp(u)))[a]
7 do
8 |  avanzar(it)
9 end
10 agRegla(actualAdelante(it).aR, u) ;                              // O(cantBits(u))[b]
```

[a] El ciclo se ejecuta tantas veces como la cantidad de versiones menores a la de la regla que estoy agregando haya en las tuplas de e.reglas. Por lo tanto, al no haber versiones repetidas, como máximo se va a recorrer version(dirIp(u)) veces la secuencia

[b] Este orden de complejidad temporal se obtiene de la interfaz del Módulo ArbolDeReglas

*i*agEvento(inout e: estr, in m: evento)

```
1  Complejidad: O(a)[a]
2  var i: interfaz
3  var tc: nat
4  var evento_prev,evento_post: evento
5  var it: itOrd(secuOrdSinRep(evento))
6  i ← m.inter ;                                                          // O(1)
7  tc ← (e.status_inter[i]).tiempoCaida ;                                 // O(1)
8  insertarOrdenadoDesdeAtras(m, (e.status_inter[i]).eventos) ;           // O(a)
9  it ← crearIt((e.status_inter[i]).eventos) ;                            // O(1)
10 if (actualAtras(it) = m) ∧ tieneAnterior?(it));                        // O(1)
11 then
12 |  retroceder(it) ;                                                   // O(1)
13 |  if (actualAtras(it).estaCaida?);                                   // O(1)
14 |  then
15 |  |  tc ← tc + (m.timestamp - actualAtras(it).timestamp);            // O(1)
16 |  end
17 end
18 else
19 |  if (tieneAnterior(it)?);                                           // O(1)
20 |  then
21 |  |  while (actualAtras(it) ≠ m);                                    // O(a)[b]
22 |  |  do
23 |  |  |  evento_post ← actualAtras(it) ;                              // O(1)
24 |  |  |  retroceder(it);                                              // O(1)
25 |  |  end
26 |  |  if (¬tieneAnterior?(it));                                       // O(1)
27 |  |  then
28 |  |  |  if (m.estaCaida?);                                          // O(1)
29 |  |  |  then
30 |  |  |  |  tc ← tc + (evento_post.timestamp - m.timestamp);         // O(1)
31 |  |  |  end
32 |  |  end
33 |  |  else
34 |  |  |  retroceder(it);                                             // O(1)
35 |  |  |  evento_prev ← actualAtras(it);                              // O(1)
36 |  |  |  if (m.estaCaida? ∧ ¬evento_prev.estaCaida?);                // O(1)
37 |  |  |  then
38 |  |  |  |  tc ← tc + (evento_post.timestamp - m.timestamp);         // O(1)
39 |  |  |  end
40 |  |  |  if (¬m.estaCaida? ∧ evento_prev.estaCaida?);                // O(1)
41 |  |  |  then
42 |  |  |  |  tc ← tc - (evento_post.timestamp - m.timestamp);         // O(1)
43 |  |  |  end
44 |  |  end
45 |  end
46 end
47 (e.status_inter[i]).tiempoCaida ← tc;                                 // O(1)
```

[a] a = longitud(e.status_inter[i].eventos) - posicionRelativa(e.status_inter[i].eventos,m)

[b] El ciclo se ejecuta a lo sumo "a" veces, dado que la secuencia se recorre de atrás hacia adelante. Es decir, que el costo temporal del ciclo es igual a la cantidad de eventos en la secuencia con timestamp mayor que el del evento que quiero agregar

*i*versiones(in e: estr) ⟶ res: conj(version)

```
1  Complejidad: O(1)
2  res ← e.versiones;                                                    // O(1)
```

*i*interfaces(in e: estr) ⟶ res: nat

```
1  Complejidad: O(1)
2  res ← e.cantInterfaces ;                                              // O(1)
```

*i*enrutar(inout e: estr, in d: dirIp) ⟶ res: RespuestaDir

```
1  Complejidad: O(version(d))
2  if (¬ pertenece(version(d),e.versiones));
   // O(posicionRelativa(e.versiones,version(v)))[a]
3  then
4  |  res ← direccionNoSoportada();                                      // O(1)
5  else
6  |  var it: itOrd(secuOrdSinRep(tupla⟨version, abr⟩))
7  |  var i: interfaz
8  |  it ← crearIt(e.reglas) ;                                           // O(1)
9  |  while (actualAdelante(it).version ≠ version(d)) ;                  // O(version(d))[b]
10 |  do
11 |  |  avanzar(it);                                                    // O(1)
12 |  end
13 |  if (¬ tieneRegla(actualAdelante(it).aR,d)) ;                       // O(version(d))[c]
14 |  then
15 |  |  res ← interfazDeSalidaNoEncontrada();                          // O(1)
16 |  else
17 |  |  i ← interfazDeSalida(actualAdelante(it).aR,d) ;                 // O(version(d))[d]
18 |  |  if (estaCaida?(e,i));                                          // O(1)
19 |  |  then
20 |  |  |  res ← interfazDeSalidaCaida(i);                             // O(1)
21 |  |  else
22 |  |  |  res ← salidaPorInterfaz(i);                                 // O(1)
23 |  |  end
24 |  end
25 end
```

[a] Cumple esta complejidad por la informacíon que aparece en la interfaz Conjunto

[b] La complejidad se da porque avanzamos sobre la secuencia de versiones hasta encontrar las reglas que corresponden con la versión de la dirIp y éstas están ordenadas

[c] Cumple esta complejidad por la informacíon que aparece en la interfaz de ArbolDeReglas

[d] Cumple esta complejidad por la informacíon que aparece en la interfaz de ArbolDeReglas

*i*eventos(in e: estr, in i: interfaz) ⟶ res: secuOrdSinRep(evento)

```
1  Complejidad: O(1)
2  res ← (e.status_inter[i]).eventos;                                    // O(1)
```

*i*estaCaida?(in e: estr, in i: interfaz) $\longrightarrow$ res: bool

```
1 Complejidad: O(1)
2 res ← false ;                                                    // O(1)
3 if (¬ vacia?((e.status_inter[i]).eventos));                      // O(1)
4 then
5 |   var it: itOrd(secuOrdSinRep(tupla ⟨nat, evento⟩))
6 |   it ← crearIt((e.status_inter[i]).eventos) ;                  // O(1)
7 |   res → actualAdelante(it).estaCaida? ;                        // O(1)
8 end
```

*i*tiempoCaida(in e: estr, in i: interfaz) $\longrightarrow$ res: nat

```
1 Complejidad: O(1)
2 res ← (e.status_inter[i]).tiempoCaida ;                          // O(1)
```

Modulo SecuenciaOrdSinRep(α,$<_\alpha$,$=_\alpha$)

Interfaz

interfaz *SecuenciaOrdSinRep(α)*
se explica con la especificación de *Secuencia(α) extendido, itOrd(α)*
usa interfaces BOOL, α, ITORD(α)
genero *secuOrdSinRep(α), itOrd(α)*

operaciones

<>()$\longrightarrow$ res: secuOrdSinRep(α) (O(1))
{true}
{$res =_{\text{obs}} <>$}

insertarOrdenadoDesdeAdelante(in a: α, inout s: secuOrdSinRep(α)) (O(posicionRelativa(s,a)))
{$s =_{\text{obs}} s_0 \wedge \neg esta?(a, s)$}
{$s =_{\text{obs}}$ agregarOrdenado(a,s_0)}

insertarOrdenadoDesdeAtras(in a: α, inout s: secuOrdSinRep(α)) (O(longitud(s) - posicionRelativa(s,a)))
{$s =_{\text{obs}} s_0 \wedge \neg esta?(a, s)$}
{$s =_{\text{obs}}$ agregarOrdenado(a,s_0)}

vacia?(in s: secuOrdSinRep(α))$\longrightarrow$ res: bool (O(1))
{true}
{$res =_{\text{obs}}$ vacia?(a,s)}

esta?(in a: α, in s: secuOrdSinRep(α))$\longrightarrow$ res: bool (O(posicionRelativa(s,a)))
{true}
{$res =_{\text{obs}}$ esta?(a,s)}

crearIt(in s: secuOrdSinRep(α))$\longrightarrow$ res: itOrd(α) (O(1))
{true}
{$res =_{\text{obs}}$ crearIt(s)}

actualAdelante(in it: itOrd(α))$\longrightarrow$ res: α (O(1))
{$\neg$vacia?(verSecuSuby(it))}
{res=$_{\text{obs}}$prim(verSecuSuby(it))}

actualAtras(in it: itOrd(α))$\longrightarrow$ res: α (O(1))
{$\neg$vacia?(verSecuSuby(it))}
{res=$_{\text{obs}}$ult(verSecuSuby(it))}

tieneProximo?(in it: itOrd(α))$\longrightarrow$ res: bool (O(1))
{$\neg$vacia?(verSecuSuby(it))}
{$res =_{\text{obs}} \neg$vacia?(fin(verSecuSuby(it)))}

tieneAnterior?(in it: itOrd(α))$\longrightarrow$ res: bool (O(1))
{$\neg$vacia?(verSecuSuby(it))}
{$res =_{\text{obs}}$ $\neg$vacia?(com(verSecuSuby(it)))}

avanzar(inout it: itOrd(α)) (O(1))
{$it =_{\text{obs}} it_0 \wedge \neg$vacia?(fin(verSecuSuby(it)))}
{verSecuSuby(it) $=_{\text{obs}}$ fin(verSecuSuby(it_0))}

retroceder(inout it: itOrd(α)) (O(1))
{$it =_{\text{obs}} it_0 \wedge \neg$vacia?(com(verSecuSuby(it)))}
{verSecuSuby(it) $=_{\text{obs}}$ com(verSecuSuby(it_0))}

Representación

secuOrdSinRep(α) **se representa con** *estrSecuOrdSinRep*
donde *estrSecuOrdSinRep* **es** *tupla* $\langle$ *prim: puntero(nodo), ult: puntero(nodo)* $\rangle$
donde *nodo* **es** *tupla* $\langle$ *dato: α, proximo: puntero(nodo), anterior: puntero(nodo)*$\rangle$

itOrd(α) **se representa con** *estrIterOrd*
donde *estrIterOrd* **es** *tupla* $\langle$ *prim: puntero(nodo), ult: puntero(nodo)* $\rangle$

Invariante de Representación

Rep : estrSecuOrdSinRep $\longrightarrow$ bool

($\forall$ e: estrSecuOrdSinRep) Rep(e) = No hay ciclos en la secuencia $\wedge$ La secuencia está ordenada $\wedge$ No hay repetidos en la secuencia $\wedge$ Recorriendo desde el primero de la secuencia se puede llegar al ultimo y viceversa.

Rep : estrIterOrd $\longrightarrow$ bool

($\forall$ e: estrIterOrd) Rep(e) = No hay ciclos en la secuencia $\wedge$ La secuencia está ordenada $\wedge$ No hay repetidos en la secuencia $\wedge$ Recorriendo desde el primero de la secuencia se puede llegar al ultimo y viceversa.

Función de Abstracción

Abs : estrSecuOrdSinRep e $\longrightarrow$ secuOrdSinRep(α) Rep(e)

($\forall$ e: estrSecuOrdSinRep) Abs(e) = s: secuOrdSinRep(α)/
(vacia?(s) $\equiv$ (e.prim == NULL)$\wedge$(e.ult == NULL)) $\wedge_{\text{L}}$ $\neg$vacia?(s) $\Rightarrow_{\text{L}}$ ((prim(s) $\equiv$ (e.prim $\rightarrow$ dato)) $\wedge$ (fin(s) $\equiv$ (final(e)))

funciones auxiliares

final : estr $\longrightarrow$ estr

final(e) $\equiv$ $\langle$(e.prim)$\rightarrow$ proximo, e.ult$\rangle$

Abs : estrIterOrd e $\longrightarrow$ itOrd(α) Rep(e)

($\forall$ e: estrIterOrd) Abs(e) = it: itOrdRef(α)/ verSecuSuby(it) $\equiv$ $Abs_{estrSecuOrdSinRep}$(e)

Algoritmos

i<>() ⟶ res:estrSecuOrdSinRep

```
1  Complejidad: O(1)
2  res ← ⟨ NULL, NULL ⟩ ;                                                    // O(1)
```

$i$insertarOrdenadoDesdeAdelante(in elem: α, inout e: estrSecuOrdSinRep)

```
1  Complejidad: O(posicionRelativa(e,elem))
2  var n: nodo
3  n.dato ← elem ;                                                           // O(1)
4  if (¬ vacia?(e));                                                         // O(1)
5  then
6      if ((*e.prim).dato >α elem ∨ (*e.ult).dato <α elem);                  // O(1)
7      then
8          if ((*e.prim).dato >α elem);                                      // O(1)
9          then
10             (*e.prim).anterior ← &n;                                      // O(1)
11             n.anterior ← NULL;                                            // O(1)
12             n.proximo ← e.prim;                                           // O(1)
13             e.prim ← &n;                                                  // O(1)
14         else
15             if ((*e.prim).dato <α elem);                                  // O(1)
16             then
17                 (*e.ult).proximo ← &n;                                    // O(1)
18                 n.anterior ← e.ult;                                       // O(1)
19                 n.proximo ← NULL;                                         // O(1)
20                 e.ult ← &n;                                               // O(1)
21             end
22         end
23     else
24         var aux: puntero(nodo);                                           // O(1)
25         aux ← e.prim;                                                     // O(1)
26         while ((*(*aux).proximo).dato <α elem);      // O(posicionRelativa(e,elem))[a]
27         do
28             aux ← (*aux).proximo;                                         // O(1)
29         end
30         (*(*aux).proximo).anterior ← &n;                                  // O(1)
31         n.proximo ← (*aux).proximo;                                       // O(1)
32         n.anterior ← aux;                                                 // O(1)
33         (*aux).proximo ← &n;                                              // O(1)
34     end
35 else
36     e.prim ← &n;                                                          // O(1)
37     e.ult ← &n;                                                           // O(1)
38     n.anterior ← NULL;                                                    // O(1)
39     n.proximo ← NULL;                                                     // O(1)
40 end
```

[a] El ciclo se ejecuta tantas veces como la cantidad de elementos menores al que estoy agregando haya en la secuencia. Por lo tanto, al no haber elementos repetidos, como máximo se va a recorrer posicionRelativa(e,elem) veces

*i*insertarOrdenadoDesdeAtras(in elem: α, inout e: estrSecuOrdSinRep)

```
1  Complejidad: O(a)[a]
2  var n: nodo
3  n.dato ← elem ;                                                    // O(1)
4  if (¬ vacia?(e));                                                  // O(1)
5  then
6      if ((*e.prim).dato >α elem ∨ (*e.ult).dato <α elem);           // O(1)
7      then
8          if ((*e.prim).dato >α elem);                               // O(1)
9          then
10             (*e.prim).anterior ← &n;                               // O(1)
11             n.anterior ← NULL;                                     // O(1)
12             n.proximo ← e.prim;                                    // O(1)
13             e.prim ← &n;                                           // O(1)
14         else
15             if ((*e.prim).dato <α elem);                           // O(1)
16             then
17                 (*e.ult).proximo ← &n;                             // O(1)
18                 n.anterior ← e.ult;                                // O(1)
19                 n.proximo ← NULL;                                  // O(1)
20                 e.ult ← &n;                                        // O(1)
21             end
22         end
23     else
24         var aux: puntero(nodo);                                    // O(1)
25         aux ← e.ult;                                               // O(1)
26         while ((*(*aux).anterior).dato >α elem);                   // O(a)[b]
27         do
28             aux ← (*aux).anterior;                                 // O(1)
29         end
30         (*(*aux).anterior).proximo ← &n;                           // O(1)
31         n.anterior ← (*aux).anterior;                              // O(1)
32         n.proximo ← aux;                                           // O(1)
33         (*aux).anterior ← &n;                                      // O(1)
34     end
35 else
36     e.prim ← &n;                                                   // O(1)
37     e.ult ← &n;                                                    // O(1)
38     n.anterior ← NULL;                                             // O(1)
39     n.proximo ← NULL;                                              // O(1)
40 end
```

[a] a = longitud(e) - posicionRelativa(e,elem)

[b] El ciclo se ejecuta tantas veces como la cantidad de elementos mayores al que estoy agregando haya en la secuencia. Por lo tanto, al no haber elementos repetidos, como máximo se va a recorrer longitud(e) - posicionRelativa(e,elem) veces

*i*vacia?(in e: estrSecuOrdSinRep) $\longrightarrow$ res: bool

```
1  Complejidad: O(1)
2  res ← (e.prim == NULL)
```

$i$esta?(in a: α, e: estrSecuOrdSinRep) $\longrightarrow$ res: bool

```
1  Complejidad: O(posicionRelativa(e,a))
2  res ← false ;                                                          // O(1)
3  if (¬vacia?(e)) ;                                                      // O(1)
4  then
5  |  var it: itOrd(secuOrdSinRep(α))
6  |  it ← crearIt(e) ;                                                  // O(1)
7  |  while (tieneProximo(it)? ∧ actualAdelante(it) <α a);   // O(posicionRelativa(e,a))[a]
8  |  do
9  |  |  avanzar(it);                                                    // O(1)
10 |  end
11 |  if (a =α actualAdelante(it));                                      // O(1)
12 |  then
13 |  |  res ← true;                                                     // O(1)
14 |  end
15 end
```

[a] La secuencia se recorre hasta que se encuentra al elemento buscado, o se encuentre uno mayor o no queden más elementos por recorrer. Por esto es que la complejidad del peor caso es posicionRelativa(e,a)

$i$crearIT(in e: estrSecuOrdSinRep) $\longrightarrow$ res: estrIterOrd

```
1  Complejidad: O(1)
2  res ← e
```

$i$actualAdelante(in it: estrIterOrd) $\longrightarrow$ res: α

```
1  Complejidad: O(1)
2  res ← ((it.prim) → dato)
```

$i$actualAtras(in it: estrIterOrd) $\longrightarrow$ res: α

```
1  Complejidad: O(1)
2  res ← ((it.ult) → dato)
```

$i$tieneProximo?(in it: estrIterOrd) $\longrightarrow$ res: bool

```
1  Complejidad: O(1)
2  res ← ((it.prim) → proximo ≠ NULL)
```

$i$tieneAnterior?(in it: estrIterOrd) $\longrightarrow$ res: bool

```
1  Complejidad: O(1)
2  res ← ((it.ult) → anterior ≠ NULL)
```

$i$avanzar(inout it: estrIterOrd)

```
1  Complejidad: O(1)
2  it.prim ← ((it.prim) → proximo)
```

*i*retroceder(inout it: estrIterOrd)

```
1 Complejidad: O(1)
2 it.ult ← ((it.ult) → anterior)
```

Modulo Conjunto(α)

Interfaz

interfaz *Conjunto(α)*
se explica con la especificación de *Conjunto(α)*
usa interfaces BOOL, NAT
genero *conj(α)*

operaciones

$Vacio() \longrightarrow res$: conj($\alpha$) (O(1))
{true}
$\{res =_{\text{obs}} \phi()\}$

agregar(in a: α, inout C: conj(α)) (O(posicionRelativa(C,a)))
$\{C =_{\text{obs}} C_0\}$
$\{C =_{\text{obs}} \text{Ag}(a,C_0)\}$

pertenece(in a: α, in C: conj(α))$\longrightarrow res$: bool (O(posicionRelativa(C,a)))
{true}
$\{res =_{\text{obs}} a \in C\}$

Representación

conjunto(α) **se representa con** *estrConj*
donde *estrConj* **es** *SecuOrdSinRep(α)*

Invariante de Representación

Rep : estrConj $\longrightarrow$ bool

($\forall$ e: estrConj) Rep(e) = true

Función de Abstracción

Abs : estrConj e $\longrightarrow$ conj(α) Rep(e)

($\forall$ e: estrConj) Abs(e) = C: conj(α)/$(\forall a : \alpha)(a \in C =_{\text{obs}} esta?(a, e))$

Algoritmos

iVacio()$\longrightarrow$ *res*: SecuOrdSinRep(α)

```
1 Complejidad: O(1)
2 res ←<>
```

iagregar(in a: α, inout s: SecuOrdSinRep(α))

```
1 Complejidad: O(posicionRelativa(s,a))
2 if ¬ esta?(a,s);                                  // O(posicionRelativa(s,a))[a]
3 then
4 |  insertarOrdenadoDesdeAdelante(a,s);            // O(posicionRelativa(s,a))[b]
5 end
```

[a]Esta complejidad se cumple por la información que brinda la interfaz de SecuOrdSinRep
[b]Esta complejidad se cumple por la información que brinda la interfaz de SecuOrdSinRep

ipertenece(in a: α, in s: SecuOrdSinRep(α))

```
1 Complejidad: O(posicionRelativa(s,a))[a]
2 res← esta?(a,s)
```

[a]Esta complejidad se cumple por la información que brinda la interfaz de SecuOrdSinRep

Modulo ArbolDeReglas

Interfaz

interfaz *arbolDeReglas*
se explica con la especificación de *arbolDeReglas*
usa interfaces BOOL, NAT, DIRIP,REGLADIR
genero *abr*

operaciones

$Crear() \longrightarrow res$: abr (O(1))
{true}
$\{res =_{\text{obs}} nuevo()\}$

$agRegla$(inout a: abr, in r: regla) (O(version(r)))
$\{a =_{\text{obs}} a_0\}$
$\{a =_{\text{obs}} agRegla(a_0, r)\}$

$interfazDeSalida$(in a: abr, in d: dirIp) $\longrightarrow res$: nat (O(version(d))))
$\{tieneRegla?(a, d)\}$
$\{res =_{\text{obs}} interfazDeSalida(a, d)\}$

Representación

arbolDeReglas **se representa con** *puntero(estr)*
donde *estr* **es**
tupla⟨ inter: puntero(interfaz)
$\times$ *izq: puntero(estr)*
$\times$ *der: puntero(estr) ⟩*

Invariante de Representación

Rep : puntero(estr) $\longrightarrow$ boolean

($\forall$ p: puntero(estr)) Rep(p) = true $\Leftrightarrow$ no existen ciclos

Función de Abstracción

Abs : puntero(estr) p $\longrightarrow$ abr Rep(p)

($\forall$ p: puntero(estr)) Abs(p) = a: abr/

($\forall$ d: dirIp) ((tieneRegla(a,d) = existeRegla(p,pasarABits(d),0) $\wedge_{\mathrm{L}}$

(tieneRegla(a,d) $\Rightarrow_{\mathrm{L}}$ (interfazDeSalida(a,d) = dameSalida(p,pasarABits(d),0,null)))))

existeRegla : puntero(estr) p $\times$ arregloDeBool $\times$ nat $\longrightarrow$ bool
Rep(p)

```
existeRegla(p,a,n)  ≡ if n = tam(a) ∨ p = null then false
                      else if (*p).inter = null then
                                  if a[n] = true then existeRegla((*e).der,a,n+1)
                                   else existeRegla((*e).izq,a,n+1)
                                   fi
                      else true
                      fi
```

dameSalida : puntero(estr) p $\times$ arregloDeBool a $\times$ nat $\times$ puntero(estr) $\longrightarrow$ interfaz
Rep(p) $\wedge_{\mathrm{L}}$ existeRegla(p,a,0)

```
dameSalida(p,a,n,pi) ≡ if n = tam(a) ∨ p = null then *pi
                       else if a[n] = true then
                                  if pi ≠ null then dameSalida((*p).der,a,n+1,pi)
                                   else dameSalida((*p).der,a,n+1,(*p).int)
                                   fi
                       else
                                  if pi ≠ null then dameSalida((*p).izq,a,n+1,pi)
                                   else dameSalida((*p).izq,a,n+1,(*p).int)
                                   fi
                       fi
```

Aclaración: dameSalida funciona dado que asumimos que el árbol se poda cada vez que agregamos una regla como lo indica el algorítmo

Algoritmos

$i$crear() $\longrightarrow$ res : puntero(estr)

```
1 Complejidad: O(1)
2 res ← NULL
```

Algorithm 1: crear

*i*agRegla(inout p : puntero(estr),in r : regla)

```
 1 Complejidad: O(version(dirIp(r)))
 2 var i : int
 3 var pAux : puntero(estr)
 4 var a : arreglo_dimensionable de bool
 5 var x : Interfaz
 6 a ← CrearArreglo(version(dirIp(r))*8) ;                    // O(version(dirIp(r)))
 7 if p = NULL ;                                               // O(1)
 8 then
 9 |  p ← & ⟨NULL,NULL,NULL⟩ ;                                 // O(1)
10 end
11 a ← pasarABits(dirIP(r)) ;                                  // O(version(dirIp(r)))
12 pAux ← p ;                                                  // O(1)
13 i ← 0 ;                                                     // O(1)
14 while i ≠ cantBits(r) ;                                     // O(cantBits(r))
15 do
16 |  if a[i] = true ;                                         // O(1)
17 |  then
18 |  |  if (*pAux).der = NULL ;                               // O(1)
19 |  |  then
20 |  |  |  (*pAux).der ← & ⟨NULL,NULL,NULL⟩ ;                 // O(1)
21 |  |  end
22 |  |  pAux ← (*pAux).der ;                                  // O(1)
23 |  else
24 |  |  if (*pAux).izq = NULL ;                               // O(1)
25 |  |  then
26 |  |  |  (*pAux).izq ← & ⟨NULL,NULL,NULL⟩ ;                 // O(1)
27 |  |  end
28 |  |  pAux ← (*pAux).izq ;                                  // O(1)
29 |  end
30 |  i ← i + 1 ;                                             // O(1)
31 end
32 if (*pAux).inter = NULL ;                                   // O(1)
33 then
34 |  x ← inter(r) ;                                          // O(1)
35 |  (*pAux).inter ← &x ;                                     // O(1)
36 else
37 |  *((*pAux).inter) ← inter(r);                            // O(1)
38 end
39 (*pAux).der ← NULL ;                                       // O(1)
40 (*pAux).izq ← NULL ;                                       // O(1)[a]
```

[a] En estas lineas podamos el arbol sin pensar el la etapa implementativa ya que sería un desperdicio de memoria. En ese momento se resolvera de alguna manera que emule la poda.

iInterfazDeSalida(in p : puntero(estr),in d : dirIp) $\longrightarrow$ res : interfaz

```
1  Complejidad: O(version(d))
2  var i : int
3  var pRes: puntero(interfaz)
4  var pResProvisorio: puntero(interfaz)
5  var pAux puntero(estr)
6  var dBits: arreglo dinamico de bool
7  pRes ← NULL ;                                              // O(1)
8  dBits ← pasarABits(d) ;                                    // O(version(d))
9  pResProvisorio ← NULL ;                                    // O(1)
10 pAux ← p ;                                                 // O(1)
11 i ← 0 ;                                                    // O(1)
12 while pRes = NULL ;                                        // O(version(d))[a]
13 do
14 |  if (pAux→inter) ≠ NULL ;                                // O(1)
15 |  then
16 |  |  pResProvisorio ← (pAux → inter) ;                   // O(1)
17 |  end
18 |  if dBits[i] = true ;                                   // O(1)
19 |  then
20 |  |  if (pAux→der) = NULL ;                              // O(1)
21 |  |  then
22 |  |  |  pRes ← pResProvisorio ;                         // O(1)
23 |  |  else
24 |  |  |  pAux ← (pAux→der) ;                             // O(1)
25 |  |  end
26 |  else
27 |  |  if (pAux→izq) = NULL ;                              // O(1)
28 |  |  then
29 |  |  |  pRes ← pResProvisorio ;                         // O(1)
30 |  |  else
31 |  |  |  pAux ← (pAux→der) ;                             // O(1)
32 |  |  end
33 |  end
34 |  i ← i + 1 ;                                           // O(1)
35 end
36 res ← *(pRes) ;                                          // O(1)
```

Algorithm 2: interfazDeSalida

[a] Esta complejidad se obtiene ya que se recorre el árbol hasta como mucho 8*version(d) y esto es O(version(d))

*i*tieneRegla(in p : puntero(estr),in d : dirIp) $\longrightarrow$ res : bool

```
1  Complejidad: O(version(d))
2  var i : int
3  var pAux : puntero(estr)
4  var sePuedeSeguir : bool
5  var dBits: arreglo dinamico de bool
6  pAux ← p ;                                                    // O(1)
7  dBits ← pasarABits(d) ;                                       // O(version(d))
8  i ← 0 ;                                                       // O(1)
9  sePuedeSeguir ← true ;                                        // O(1)
10 res ← false ;                                                 // O(1)
11 if pAux = NULL ;                                              // O(1)
12 then
13 |  res ← false ;                                              // O(1)
14 |  sePuedeSeguir ← false ;                                    // O(1)
15 end
16 while res = false ∧ sePuedeSeguir = true ;                    // O(version(d))[a]
17 do
18 |  if dBits[i] = true ;                                       // O(1)
19 |  then
20 |  |  if (pAux→der) = NULL ;                                  // O(1)
21 |  |  then
22 |  |  |  sePuedeSeguir ← false ;                             // O(1)
23 |  |  else
24 |  |  |  pAux ← (pAux→der) ;                                 // O(1)
25 |  |  end
26 |  else
27 |  |  if (pAux→izq) = NULL ;                                  // O(1)
28 |  |  then
29 |  |  |  sePuedeSeguir ← false ;                             // O(1)
30 |  |  else
31 |  |  |  pAux ← (pAux→izq) ;                                 // O(1)
32 |  |  end
33 |  end
34 |  i ← i + 1 ;                                                // O(1)
35 |  if (pAux→inter) ≠ NULL ;                                   // O(1)
36 |  then
37 |  |  res ← true ;                                           // O(1)
38 |  end
39 end
```

[a] Esta complejidad se obtiene ya que se recorre el arbol hasta como mucho 8*version(d) y esto es O(version(d))

$i$pasarABits(in d : dirIp) $\longrightarrow$ res : arreglo_dinamico de bool

```
1  Complejidad: O(version(d))
2  var a : arreglo_dinamico de bool
3  var i, j, aux : nat
4  a ← crearArrego(8*version(d)) ;                          // O(version(d))
5  i ← 7 ;                                                  // O(1)
6  j ← 0 ;                                                  // O(1)
7  while (j < version(d)) ;                                 // O(version(d))[a]
8  do
9  |  aux ← d[j] ;                                         // O(1)
10 |  while (i ≥ 8*j) ;                                    // O(1)[b]
11 |  do
12 |  |  if aux mod 2 = 1 ;                                // O(1)
13 |  |  then
14 |  |  |  a[i] ← true ;                                 // O(1)
15 |  |  else
16 |  |  |  a[i] ← false ;                                // O(1)
17 |  |  end
18 |  |  aux ← aux / 2 ;                                  // O(1)
19 |  |  i ← i − 1 ;                                      // O(1)
20 |  end
21 |  i ← i + 15 ;                                        // O(1)
22 |  j ← j + 1 ;                                         // O(1)
23 end
24 res ← a ;                                              // O(1)
```

Algorithm 3: pasarABits

[a] Esto se ejecuta version(d) veces. Esto también podría haberse implementado con un "for" en vez de con un "while". Por cada iteración de "j" se itera 8 veces con "i", por lo que la complejidad total es O(8*version(d)), pero asintóticamente es O(version(d))

[b] Esto se ejecuta 8 veces, porque cuando se entra aquí por primera vez vale que i = j+7. Esto también podría haberse implementado con un "for" en vez de con un "while"

TAD (SecuOrdSinRep(α),$<_\alpha$,$=_\alpha$)

extiende Secu(α)

géneros secuOrdSinRep(α)

exporta genero, generadores, observadores básicos

generadores

agregarOrdenado	: α a × secuOrdSinRep(α) s	⟶	secuOrdSinRep(α)	(¬esta?(a,s))

otras operaciones

ordenada	: secuOrdSinRep(α)	⟶	bool	
posicionRelativa	: secuOrdSinRep(α) × α	⟶	nat	
com	: secuOrdSinRep(α) s	⟶	secuOrdSinRep(α)	¬vacia?(s)

axiomas (∀ s: secu(α))(∀ a: α)

vacia?(agregarOrdenado(a, s))	≡	*false*
prim(agregarOrdenado(a, s))	≡	**if** (vacia?(s)) **then** a **else** min(a,prim(s))
fin(agregarOrdenado(a, s))	≡	**if** (vacia?(s)) $\vee_L$ a $<_\alpha$ prim(s)) **then** s **else** agregarOrdenado(a, fin(s)))
ordenada(<>)	≡	true
ordenada(agregarOrdenado(a,s))	≡	(a $<_\alpha$ prim(s))) $\wedge$ ordenada(s))
com(s)	≡	**if** (vacia?(fin(s))) **then** **else** agregarOrdenado(prim(s),com(fin(s)))
posicionRelativa(s,a)	≡	**if** (vacia?(s) $\vee_L$a $\leq_\alpha$ prim(s)) **then** 0 **else** 1 + posicionRelativa(fin(s),a)

Fin TAD

TAD ItOrd(α)

Igualdad Observacional

(∀ it ,it': itOrd(α)) ($it =_{\text{obs}} it' \Longleftrightarrow (VerSecuSuby(it) =_{\text{secu}(\alpha)} VerSecuSuby(it'))$)

géneros itOrd(α)

usa Secu(α)

exporta género, generadores, observadores básicos

generadores

CrearIt	: secu(α)	$\longrightarrow$	itOrd(α)

observadores básicos

VerSecuSuby	: itOrd(α)	$\longrightarrow$	secu(α)

axiomas (∀ s: secu(α))

VerSecuSuby(CrearIt(s)) $\equiv$ s

Fin TAD

TAD ArbolDeReglas

Igualdad Observacional

$$(\forall \text{ aR ,aR': itOrd}(\alpha)) \left(\begin{array}{c} aR =_{\text{obs}} aR' \Longleftrightarrow (\forall d : dirIp)(tieneRegla(aR,d) =_{\text{obs}} tieneRegla(aR',d) \wedge_{\text{L}} \\ tieneRegla(aR,d) \Rightarrow_{\text{L}} interfazDeSalida(aR,d) =_{\text{obs}} interfazDeSalida(aR',d)) \end{array} \right)$$

géneros arbolDeReglas

usa Bool, Nat, ReglaDir, DirIp

exporta género, generadores, observadores básicos

generadores

nuevo	:	$\longrightarrow$	ArbolDeReglas
agRegla	: ArbolDeReglas $\times$ Regla	$\longrightarrow$	ArbolDeReglas

observadores básicos

interfazDeSalida	: ArbolDeReglas a $\times$ dirIp d	$\longrightarrow$	nat	tieneRegla(a,d)
tieneRegla	: ArbolDeReglas a $\times$ dirIp d	$\longrightarrow$	bool	

otras operaciones

primerosBitsIguales? : dirIP $d1$ $\times$ dirIP $d2$ $\times$ nat $\longrightarrow$ bool
((version(d1) = (version(d2)) $\wedge$ (version(d1)*8 $\geq$ n))

axiomas (∀ a: arbolDeReglas)(∀ r: regla)(∀ d,d': dirIp)

interfazDeSalida(agRegla(a,r),d)	$\equiv$	If primerosBitsIguales?(dirIp(r),d,cantBits(r)) then inter(r) else interfazDeSalida(a,d)
tieneRegla(nuevo(),d)	$\equiv$	false
tieneRegla(agRegla(a,r),d)	$\equiv$	If primerosBitsIguales?(dirIp(r),d, cantBits(r)) then true else tieneRegla(a,d)
primerosBitsIguales?(d,d',n)	$\equiv$	**if** (n=0) **then** *true* **else** pasarA8Bits(d[n/8])[n mod 8] = pasarA8Bits(d'[n/8])[n mod 8] $\wedge$ primerosBitsIguales?(d,d',pred(n))

Fin TAD

TAD CONJUNTO(α)

extiende Conjunto(α)

géneros conj(α)

otras operaciones

posicionRelativa : conj(α) $\times$ α $\longrightarrow$ nat

axiomas ($\forall$ c: conj(α))($\forall$ a: α)

posicionRelativa(c,a) $\equiv$ **if** ($\emptyset$?(c)) **then** 0
else if (a $\leq_\alpha$ dameUno(c)) **then** posicionRelativa(sinUno(c),a)
else 1 + posicionRelativa(sinUno(c),a)

Fin TAD