Algoritmos y Estructuras de Datos II

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

De los creadores de sacarCompu...

Trabajo práctico 2

Diseño - DCNet

Grupo 11

Integrante	LU	Correo electrónico
Frizzo, Franco	013/14	francofrizzo@gmail.com
Martínez, Manuela	160/14	martinez.manuela.22@gmail.com
Rabinowicz, Lucía	105/14	lu.rabinowicz@gmail.com
Weber, Andrés	923/13	herr.andyweber@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo Red

Notas preliminares

En todos los casos, al indicar las complejidades de los algoritmos, las variables que se utilizan corresponden a:

- n: Número de computadoras en la red.
- L: Longitud de nombre de computadora más largo de la red.
- I: Mayor cantidad de interfaces que tiene alguna computadora en la red en el momento.

Servicios usados: interfaz, tupla, nat, IP, lista

Interfaz

se explica con: RED, ITERADOR UNIDIRECCIONAL(COMPU)

géneros: `red`, `itRed`

Operaciones del TAD Red

INICIARRED() $\rightarrow$ *res* : `Red`
Pre $\equiv$ {}
Post $\equiv$ {res $=_{\text{obs}}$ iniciarRed()}
Complejidad: $\Theta(1)$
Descripción: Genera una nueva red sin ninguna computadora.

AGREGARCOMPU(**in/out** r : `Red`, **in** c : `compu`)
Pre $\equiv$ {$r =_{\text{obs}} r_0 \wedge (\forall c'$: compu)($c' \in$ computadoras(r) $\rightarrow$ ip(c) $\neq$ ip(c'))}
Post $\equiv$ {$r =_{\text{obs}}$ agregarCompu(r_0, c)}
Complejidad: $\Theta(I)$
Descripción: Agrega una nueva computadora a la red.

CONECTAR(**in/out** r : `Red`, **in** c_0 : `compu`, **in** i_0 : `interfaz`, **in** c_1 : `compu`, **in** i_1 : `interfaz`) $\rightarrow$ *res* : `Red`
Pre $\equiv$ {$r =_{\text{obs}} r_0 \wedge c_1 \in$ computadoras(r) $\wedge$ $c_2 \in$ computadoras(r) $\wedge$ ip(c_0) $\neq$ ip(c_1) $\wedge$ $\neg$conectadas?(r, c_0 , c_1) $\wedge$ $\neg$usaInterfaz?(r, c_0, i_0) $\wedge$ $\neg$usaInterfaz?(r, c_1, i_1)}
Post $\equiv$ {$r =_{\text{obs}}$ conectar(r_0, c_0, i_0, c_1, i_1)}
Complejidad: $\Theta(n + I)$
Descripción: Conecta la computadora c_0 con la computadora c_1 a través de las interfaces i_0 y i_1 respectivamente.

COMPUTADORAS(**in** r : `Red`) $\rightarrow$ *res* : `conj(compu)`
Pre $\equiv$ {}
Post $\equiv$ {esAlias(*res*, computadoras(r))}
Complejidad: $\Theta(1)$
Descripción: Devuelve el conjunto de todas las computadoras de la red.
Aliasing: El conjunto es devuelto por referencia.

CONECTADAS?(**in** r : `Red`, **in** c_0 : `compu`, **in** c_1 : `compu`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {$c_0 \in$ computadoras(r) $\wedge$ $c_1 \in$ computadoras(r)}
Post $\equiv$ {*res* $=_{\text{obs}}$ conectadas?(r, c_0, c_1)}
Complejidad: $\Theta(n + I)$
Descripción: Devuelve *true* si y solo si la computadora c_0 esta conectada a la computadora c_1

INTERFAZUSADA(**in** r : `Red`, **in** c_0 : `compu`, **in** c_1 : `compu`) $\rightarrow$ *res* : `interfaz`
Pre $\equiv$ {$c_0 \in$ computadoras(r) $\wedge$ $c_1 \in$ computadoras(r) $\wedge_{\text{L}}$ conectadas?(r, c_0, c_1)}
Post $\equiv$ {*res* $=_{\text{obs}}$ interfazUsada(r, c_0, c_1)}
Complejidad: $\Theta(n + I)$
Descripción: Devuelve la interfaz usada por c_0 para conectarse a c_1

VECINOS(**in** r : `Red`, **in** c : `compu`) $\rightarrow$ *res* : `conj(compu)`
Pre $\equiv$ {$c \in$ computadoras(r)}

Post $\equiv \{res =_{\text{obs}} \text{vecinos}(r, c)\}$
Complejidad: $\Theta(n + I^3)$
Descripción: Devuelve el conjunto de vecinos de la computadora c, es decir, las computadoras que tienen una conexión directa con c.
Aliasing: Devuelve el conjunto por copia.

USAINTERFAZ?(**in** r : `Red`, **in** c : `compu`, **in** i : `interfaz`) $\rightarrow$ res : `bool`
Pre $\equiv \{c \in \text{computadoras}(r)\}$
Post $\equiv \{res =_{\text{obs}} \text{usaInterfaz?}(r, c, i)\}$
Complejidad: $\Theta(n + I)$
Descripción: Devuelve *true* si y solo si la computadora c está usando la interfaz i.

CAMINOSMINIMOS(**in** r : `Red`, **in** c_0 : `compu`, **in** c_1 : `compu`) $\rightarrow$ res : `conj(secu(compu))`
Pre $\equiv \{c_0 \in \text{computadoras}(r) \land c_1 \in \text{computadoras}(r)\}$
Post $\equiv \{res =_{\text{obs}} \text{caminosMinimos}(r, c_0, c_1)\}$
Complejidad: $\Theta(n^3 \times n! \times n! + I)$
Descripción: Devuelve el conjunto de todos los caminos mńimos posibles entre c_0 y c_1. De no haber ninguno, devuelve $\emptyset$.
Aliasing: Devuelve el conjunto por copia.

HAYCAMINO?(**in** r : `Red`, **in** c_0 : `compu`, **in** c_1 : `compu`) $\rightarrow$ res : `bool`
Pre $\equiv \{c_0 \in \text{computadoras}(r) \land c_1 \in \text{computadoras}(r)\}$
Post $\equiv \{res =_{\text{obs}} \text{hayCamino?}(r, c_0, c_1)\}$
Complejidad: $\Theta(n^2 \times n!)$
Descripción: Devuelve *true* si y solo si hay al menos un camino posible entre c_0 y c_1.

CANTCOMPUS(**in** r : `Red`) $\rightarrow$ res : `nat`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \#(\text{computadoras(r)})\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve cuántas computadoras hay en la red.

COPIAR(**in** r : `Red`) $\rightarrow$ res : `Red`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} r\}$
Complejidad: $\Theta(n \times I)$
Descripción: Devuelve una copia de la red.

Representación

`red` se representa con `estrRed`

donde `estrRed` es `tupla`(*compus*: `conjunto(compu)` , *conexiones*: `dicc(IP, diccConexiones)`)

donde `diccConexiones` es `dicc(interfaz, itDicc(IP, diccConexiones))`

Rep en castellano:

1. El cardinal del conjunto de claves del diccionario Conexiones, es igual al cardinal del conjunto de compus.
2. Ninguna compu tiene vecinos repetidos.
3. Ninguna compu puede ser vecina de si misma.
4. Todos los vecinos tienen que estar en el conjunto de claves del diccionario Conexiones.
5. Si c_1 y c_2 son computadoras de la red, c_1 es vecina de c_2 si y solo si c_2 es vecina de c_1.
6. El conjunto de compus es el mismo que el que resulta de armar las computadoras con los diccionarios.

Rep : Red $\longrightarrow$ bool

Rep(e) $\equiv$

1. #claves(e.conexiones)=#(e.compus) $\wedge$
2. ($\forall c$: IP) $c \in$ claves(e.conexiones) $\Rightarrow_{\text{L}}$ sinRepetidos(JuntarSignificados(obtener(c, e.conexiones))) $\wedge$
3. ($\forall c$: IP) $c \in$ claves(e.conexiones) $\Rightarrow_{\text{L}}$ $\neg$esta?(c, JuntarSignificados(obtener(c, e.conexiones))) $\wedge$
4. ($\forall c_1$:IP) $c \in$ claves(e.conexiones) $\Rightarrow_{\text{L}}$ ($\forall c_2$:IP) esta?(c_2, JuntarSignificados(obtener(c_1, e.conexiones))) $c_2 \Rightarrow$ (e.conexiones) $\wedge_{\text{L}}$
5. ($\forall c_1$:IP) ($\forall c_2$:IP) ($c_1 \in$ claves(e.conexiones)) $\wedge$ ($c_2 \in$ claves(e.conexiones)) $\Rightarrow_{\text{L}}$ esta?(c_1, JuntarSignificados(obtener(c_2, e.conexiones))) $\Leftrightarrow$ esta?(c_2, JuntarSignificados(obtener(c_1, e.conexiones))) $\wedge$
6. ($\forall c$: IP) $c \in e$.compus $\Rightarrow \Pi_1(c) \in$ claves(e.conexiones) $\wedge_{\text{L}}$ claves(obtener($\Pi_1(c)$, e.conexiones)) $\subseteq \Pi_2(c)$

Abs : estrRed e $\longrightarrow$ Red {Rep(e)}

Abs(e) $\equiv$ (r: Red | computadoras(r) = ArmarComputadoras(e.compus) $\wedge$
($\forall c_1$: compu)(($\forall c_2$: compu) conectados?(r, c_1, c_2) = Pertenece?(e.compus, c_1, c_2) $\wedge$
InterfazUsada(r, c_1, c_2) = DevolverInterfaz(e.compus, c_1, c_2)))

ArmarComputadoras : secu(tupla(string,secu(tupla(Interfaz,ItRed)))) $\longrightarrow$ conj(compu)

ArmarComputadoras(l) $\equiv$ **if** vacia?(l) **then**
$\emptyset$
else
Ag($\langle \Pi_1(\text{prim}(l)), \text{GenerarInterfaces}(\Pi_2(\text{prim}(l)))\rangle$, ArmarComputadoras(fin($l$)))
fi

ArmarSecuencia : secu(tupla(string,secu(tupla(interfaz,itLista(compu))))) $\longrightarrow$ secu(string)

ArmarSecuencia(s) $\equiv$ **if** vacia?(s) **then** $<>$ **else** (Π_1(prim(s))) $\bullet$ ArmarSecuencia(fin(s)) **fi**

sinRepetidos : secu(string) $\longrightarrow$ bool

sinRepetidos(s) $\equiv$ #(pasarSecuAConj(s) = long(s)

pasarSecuAConj : secu(string) $\longrightarrow$ conj(string)

pasarSecuAConj(s) $\equiv$ **if** vacia?(s) **then** $\emptyset$ **else** Ag(prim(s), pasarSecuAConj(fin(s))) **fi**

GenerarInterfaces : secu(tupla(Interfaz,ItLista(estrCompu))) $\longrightarrow$ conj(Interfaz)

GenerarInterfaces(l) $\equiv$ **if** vacia?(l) **then** $\emptyset$ **else** Ag(Π_1(prim(l)), GenerarInterfaces(fin(l))) **fi**

Pertenece? : secu(tupla(string,secu(tupla(Interfaz,ItRed)))) l $\times$ compu c_1 $\times$ compu c_2 $\longrightarrow$ bool

Pertenece?(l, c_1, c_2) $\equiv$ **if** (Π_1(prim(l) = Π_1(c_1))) **then**
$\Pi_1(c_2) \in$ GenerarCompus(Π_2(prim(l)))
else
Pertenece?(fin(l), c_1, c_2)
fi

GenerarCompus : secu(tupla<Interfaz $\times$ ItLista(estrCompu)>) $\longrightarrow$ conj(string)

GenerarCompus(l) $\equiv$ **if** vacia?(l) **then** $\emptyset$ **else** Ag(Π_1(siguiente(Π_2(prim(l)))), GenerarCompus(fin(l))) **fi**

DevolverInterfaz : secu(tupla(string × secu(tupla(Interfaz × ItRed)))) l × compu c_1 × compu c_2 $\longrightarrow$ Interfaz
{Pertenece?(l, c_1, c_2}

DevolverInterfaz(l, c_1, c_2) $\equiv$ **if** (Π_1(prim(l)) = Π_1(c_1)) **then**
DevolverInterfaz$_{\text{aux}}$(Π_2(prim(l), c_2))
else
DevolverInterfaz(fin(l, c_1, c_2))
fi

DevolverInterfaz$_{\text{aux}}$: secu(tupla(Interfaz × ItRed)) l × compu c $\longrightarrow$ Interfaz

DevolverInterfaz(l, c) $\equiv$ **if** ($\Pi_1(c_2)$ = Π_1(siguiente(Π_2(prim(l))))) **then**
Π_1(prim(l))
else
DevolverInterfaz$_{\text{aux}}$(fin(l, c))
fi

Algoritmos

⚠ Algoritmo modificado

iIniciarRed() → res : estrRed	
1 res ← ⟨Vacio(), Vacio()⟩	▷ $\Theta(1)$

Complejidad: $\Theta(1)$

⚠ Algoritmo modificado

iAgregarCompu(**in/out** r: estrRed, **in** c: compu)	
1 AgregarRapido($r.compus$, c)	▷ $\Theta(1)$
2 DefinirRapido($r.conexiones$, $c.IP$, Vacio())	▷ $\Theta(1)$

Complejidad: $\Theta(1)$

⚠ Algoritmo modificado

iConectar(**in/out** r: estrRed), **in** c_1: compu, **in** i_1: interfaz, **in** c_2: compu, **in** i_2: interfaz)	
1 itDicc(IP, diccConexiones) it_1 ← CrearIt($r.conexiones$)	▷ $\Theta(1)$
2 itDicc(IP, diccConexiones) it_2 ← CrearIt($r.conexiones$)	▷ $\Theta(1)$
3 **while** SiguienteClave(it_1) ≠ $c_1.IP$ **do**	▷ $\Theta(n)$ iteraciones
4 | Avanzar(it_1)	▷ $\Theta(1)$
5 **end while**	
6 **while** SiguienteClave(it_2) ≠ $c_1.IP$ **do**	▷ $\Theta(n)$ iteraciones
7 | Avanzar(it_2)	▷ $\Theta(1)$
8 **end while**	
9 DefinirRapido(SiguienteSignificado(it_1), i_1, Copiar(it_2))	▷ $\Theta(1)$
10 DefinirRapido(SiguienteSignificado(it_2), i_2, Copiar(it_1))	▷ $\Theta(1)$

Complejidad: $\Theta(n)$

Justificación: El algoritmo tiene dos ciclos que se ejecutan $\Theta(n)$ veces, cada una con complejidad $\Theta(1)$. El resto

de las operaciones tiene complejidad $\Theta(1)$.

Algoritmo modificado

iConectadas?(**in** r : estrRed, **in** c_1 : compu, **in** c_2 : compu) $\rightarrow$ res : bool

```
1  itDicc(IP, diccConexiones) it1 ← CrearIt(r.conexiones)                          ▷ Θ(1)
2  while SiguienteClave(it1) ≠ c1.IP do                                            ▷ Θ(n) iteraciones
3  |  Avanzar(it1)                                                                ▷ Θ(1)
4  end while
5  itDicc(interfaz, itDicc(IP, diccConexiones)) it2 ← CrearIt(Significado(Siguiente(it1)))   ▷ Θ(1)
6  while HaySiguiente?(it2) ∧L SiguienteClave(SiguienteSignificado(it2)) ≠ c2.IP do   ▷ Θ(I) iteraciones
7  |  Avanzar(it2)                                                                ▷ Θ(1)
8  end while
9  res ← HaySiguiente?(it2) ∧L SiguienteClave(SiguienteSignificado(it2)) = c2.IP   ▷ Θ(1)
```

Complejidad: $\Theta(n + I)$

Justificación: El algoritmo tiene dos ciclos; uno de ellos se ejecuta $\Theta(n)$ veces, y el otro $\Theta(I)$ veces, todas ellas con complejidad $\Theta(1)$. El resto de las operaciones tiene complejidad $\Theta(1)$.

Algoritmo modificado

iInterfazUsada(**in** r : estrRed, **in** c_1 : compu, **in** c_2 : compu) $\rightarrow$ res : interfaz

```
1  itLista(estrCompu) it1 ← crearIt(r.compus)                                      ▷ Θ(1)
2  while siguiente(it1).IP ≠ c1.IP do                                              ▷ Θ(n) iteraciones
3  |  avanzar(it1)                                                                ▷ Θ(1)
4  end while
5  itLista(tupla(interfaz, itLista(estrCompu))) it2 ← crearIt(siguiente(it1).conexiones)   ▷ Θ(1)
6  while (siguiente(siguiente(it2).com)).IP ≠ c2.IP do                             ▷ Θ(I) iteraciones
7  |  avanzar(it1)                                                                ▷ Θ(1)
8  end while
9  res ← siguiente(it2).inter                                                      ▷ Θ(1)
```

Complejidad: $\Theta(n + I)$

iVecinos(**in** r : estrRed, **in** c : compu) $\rightarrow$ res : conj(compu)

```
1  res ← vacio()                                                                   ▷ Θ(1)
2  itLista(estrComp) it1 ← crearIt(r.compus)                                       ▷ Θ(1)
3  while siguiente(it1).IP ≠ c.IP do                                               ▷ Θ(n) iteraciones
4  |  avanzar(it1)                                                                ▷ Θ(1)
5  end while
6  itLista(tupla(interfaz, itLista(estrCompu))) it2 ← crearIt(siguiente(it1).conexiones)   ▷ Θ(1)
7  while haySiguiente?(it2) do                                                     ▷ Θ(n) iteraciones
8  |  if haySiguiente?(siguiente(it2).com) then                                   ▷ Θ(1)
9  |  |  agregar(res, ⟨siguiente(siguiente(it2).com).IP,
   |  |  crearConjunto(siguiente(siguiente(it2).com).conexiones)⟩)               ▷ Θ(I²)
10 |  end if
11 |  avanzar(it2)                                                                ▷ Θ(1)
12 end while
```

Complejidad: $\Theta(n + I^3)$

iCrearConjunto(**in** l: `lista(tupla(`$inter$`: interfaz,` com`: itLista(estrCompu)))`) → res : `conj(interfaz)`

```
1 nat n ← 0                                                          ▷ Θ(1)
2 res ← vacio()                                                      ▷ Θ(1)
3 while n <longitud(l) do                                            ▷ Θ(I) iteraciones
4 |  agregar(res, (l[n]).inter)                                      ▷ Θ(I)
5 |  n ← n + 1                                                       ▷ Θ(1)
6 end while
```

Descripción: Dada una lista de tupla de ⟨Interfaz,Iterador⟩ (que representa las conexiones de la computadora), devuelve el conjunto de todas las interfaces que se encuentran en ella.

Complejidad: $\Theta(I^2)$

iUsaInterfaz?(**in** r : `estrRed`, **in** c: `compu`, **in** i: `interfaz`) → res : `bool`

```
1 itLista(estrComp) it1 ← crearIt(r.compus)                          ▷ Θ(1)
2 while siguiente(it1).IP ≠ c.IP do                                  ▷ Θ(n) iteraciones
3 |  avanzar(it1)                                                    ▷ Θ(1)
4 end while
5 itLista(tupla(interfaz, itLista(estrCompu))) it2 ← crearIt(siguiente(it1).conexiones)   ▷ Θ(1)
6 while siguiente(it2).inter ≠ i do                                  ▷ Θ(I) iteraciones
7 |  avanzar(it2)                                                    ▷ Θ(1)
8 end while
9 res ← haySiguiente(siguiente(it2).com)                             ▷ Θ(1)
```

Complejidad: $\Theta(n + I)$

iCaminosMinimos(**in** r : `estrRed`, **in** c_1: `compu`, **in** c_2: `compu`) → res : `conj(lista(compu))`

```
1 res ← vacio()                                                      ▷ Θ(1)
2 if pertenece?(c2, vecinos(r, c1)) then                             ▷ Θ(I)
3 |  agregar(res, agregarAtras(agregarAtras(<>,c1), c2))             ▷ Θ(n + I)
4 else
5 |  res ← dameMinimos(Caminos(r, c1, c2, agregarAtras(<>,c1), pasarConjASecu(vecinos(r, c1))))
  |   ▷ Θ(n^3 × n! × n!)
6 end if
```

Complejidad: $\Theta(n^3 \times n! \times n! + I)$

dameMinimos(**in** c: `conj(lista(compu))`) → res : `conj(lista(compu))`

```
1 if esVacio?(c) then                                                ▷ Θ(1)
2 |  res ← vacio()                                                   ▷ Θ(1)
3 else
4 |  itConj(lista(compu)) it ← crearIt(c)                            ▷ Θ(1)
5 |  res ← dameMinimosAux(c, minimaLong(c, long(siguiente(it))))     ▷ Θ(n × n!)
6 end if
```

Descripción: Devuelve, del total de caminos posibles, solo los de longitud mínima

Complejidad: $\Theta(n \times n!)$

```
DAMEMINIMOSAUX(in c: conj(lista(compu)), in n: nat) → res : conj(lista(compu))
1  itConj(lista(compu)) it ← crearIt(c)                                  ▷ Θ(1)
2  res ← vacio()                                                         ▷ Θ(1)
3  while haySiguiente(it) do                                             ▷ Θ(n!) iteraciones
4  |  if long(siguiente(it))= n then                                     ▷ Θ(1)
5  |  |  agregar(res, siguiente(it))                                     ▷ Θ(n)
6  |  |  avanzar(it)                                                     ▷ Θ(1)
7  |  else
8  |  |  avanzar(it)                                                     ▷ Θ(1)
9  |  end if
10 end while
```

Complejidad: $\Theta(n \times n!)$

```
MINIMALONG(in c: conj(lista(compu)), in n: nat) → res : nat
1  nat i ← n                                                             ▷ Θ(1)
2  itConj(lista(compu)) it ← crearIt(c)                                  ▷ Θ(1)
3  while haySiguiente(it) do
4  |  if long(siguiente(it)) then
5  |  |  i ← longitud(siguiente(it))                                     ▷ Θ(1)
6  |  |  avanzar(it)                                                     ▷ Θ(1)
7  |  else
8  |  |  avanzar(it)                                                     ▷ Θ(1)
9  |  end if
10 end while
11 res ← i                                                               ▷ Θ(1)
```

Complejidad: $\Theta(n!)$

Justificación: Devuelve la longitud de la secuencia más chica

```
PASARCONJASECU(in c: conj(compu)) → res : secu(compu)
1  res ← vacia()                                                         ▷ Θ(1)
2  ItConj it ← crearIt(c)                                                ▷ Θ(1)
3  while haySiguiente(it) do                                             ▷ Θ(n) iteraciones
4  |  agregarAtras(res, siguiente(it))                                   ▷ Θ(I)
5  end while
```

Complejidad: $\Theta(n \times I)$

Justificación: Devuelve una secuencia que contiene a todos los elementos del conjunto pasado por parámetro

```
IHAYCAMINO?(in r: estrRed, in c1: compu, in c2: compu) → res : bool
1  res ← (¬esVacio?(iCaminosMinimos(r, c1, c2)))                         ▷ Θ(n² × n!)
```

Complejidad: $\Theta(n^2 \times n!)$

```
iCaminos(in r: estrRed, in c1: compu, in c2: compu, in l: lista(estrCompu), in vec: lista(estrCompu))
→ res : conj(lista(estrCompu))
1  if vacia?(vec) then
2  |  res ← vacia()
3  else
4  |  if último(l) = c1 then
5  |  |  res ← agregar(l, vacia())
6  |  else
7  |  |  if ¬está?(primero(vec, l)) then
8  |  |  |  res ← unión(caminos(r, c0, c1, agregarAtras(l, primero(vec)), Vecinos(r, primeros(vec))),
   |  |  |  caminos(r, c0, c1, l, fin(vec)))
9  |  |  else
10 |  |  |  res ← caminos(r, c0, c1, l, fin(vec))
11 |  |  end if
12 |  end if
13 end if
```

Descripción: Dada una red, dos compus, los vecinos de la primer compu, y una lista que usamos para guardar las computadoras por las que ya preguntamos, iteramos sobre todas las computadoras y devolvemos el conjunto de todos los caminos posibles desde la primer computadora hasta la segunda.

Complejidad: $\Theta(n^2 \times n!)$

```
iUnión(in c1: conj(lista(compu)), in c2: conj(lista(compu))) → res : conj(lista(compu))
1  res ← vacio()                                                          ▷ Θ(1)
2  if vacio?(c1) then                                                     ▷ Θ(1)
3  |  res ← c2                                                           ▷ Θ(I × n × n!)
4  else
5  |  itConj(lista(compu)) it ← crearIt(c1)                              ▷ Θ(1)
6  |  while haySiguiente(it) do                                          ▷ Θ(n)
7  |  |  Ag(siguiente(it), c2)                                           ▷ Θ(n)
8  |  |  avanzar(it)                                                     ▷ Θ(1)
9  |  end while
10 end if
```

Complejidad: $\Theta(n^2 + n \times I \times n!)$

Justificación: Devuelve la unión de dos conjuntos.

```
iEsta?(in c: compu, in l: lista(compu)) → res : bool
1  if vacia?(l) then                                                      ▷ Θ(1)
2  |  res ← false                                                        ▷ Θ(1)
3  else
4  |  ItLista(compu) it ← crearIt(l)                                     ▷ Θ(1)
5  |  while haySiguiente(it) ∧L siguiente(it) ≠ c do                     ▷ Θ(n)
6  |
7  |  |  avanzar(it)                                                     ▷ Θ(1)
8  |  end while
9  end if
10 res ← (haySiguiente(it))                                              ▷ Θ(1)
```

Descripción: Devuelve True si y solo si la compu c se encuentra en la lista l

Complejidad: $\Theta(n)$

Justificación: .

iComputadoras(**in** r : `estrRed`) $\rightarrow$ res : `conj(compu)`

```
1 res ← vacio()                                          ▷ Θ(1)
2 itRed it ← crearItRed()                                ▷ Θ(1)
3 while haySiguiente?(it) do                             ▷ Θ(n) iteraciones
4 |   agregar(res, siguiente(it))                        ▷ Θ(n + I²)
5 |   avanzar(it)                                        ▷ Θ(1)
6 end while
```

Complejidad: $\Theta(n \times (n + I^2))$

iCopiar(**in** r : : estrRed) $\rightarrow$ res : `Red`

```
1 res ← ⟨copiar(r.compus, r.cantidadCompus⟩              ▷ Θ(n × I)
```

Complejidad: $\Theta(n \times I)$

Justificación: Devuelve una copia de la Red

iCantCompus(**in** r : `Red`) $\rightarrow$ res : `nat`

```
1 res ← r.cantCompus                                     ▷ Θ(1)
```

Complejidad: $\Theta(1)$

2. Módulo Árbol binario

Notas preliminares

En todos los casos, al indicar las complejidades de los algoritmos, las variables que se utilizan corresponden a:

- n: Cantidad de nodos en el árbol binario.

Interfaz

parámetros formales

géneros α

se explica con: ÁRBOL BINARIO(α)

géneros: `ab`(α)

Operaciones del TAD Árbol binario

NIL() $\rightarrow$ *res* : `ab`(α)
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ nil}
Complejidad: $\Theta(1)$
Descripción: Crea y devuelve un árbol binario vacío.

BIN(**in** i: `ab`(α), **in** r: α, **in** d: `ab`(α)) $\rightarrow$ *res* : `ab`(α)
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ bin(i, r, d) $\wedge$ esAlias(izq(*res*), i) $\wedge$ esAlias(raiz(*res*), r) $\wedge$ esAlias(der(*res*), d)}
Complejidad: $\Theta(1)$
Descripción: Crea y devuelve un árbol binario usando los parámetros de entrada como subárbol izquierdo, raíz y subárbol derecho, respectivamente.
Aliasing: Tanto la raíz como los dos subárboles son tomados por referencia. Cualquier modificación de los mismos incide sobre el árbol binario creado.

ESNIL(**in** a: `ab`(α)) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ nil?(a)}
Complejidad: $\Theta(1)$
Descripción: Devuelve *true* si y solo si el árbol binario está vacío.

RAIZ(**in** a: `ab`(α)) $\rightarrow$ *res* : α
Pre $\equiv$ {$\neg$ nil?(a)}
Post $\equiv$ {esAlias(*res*, raiz(a))}
Complejidad: $\Theta(1)$
Descripción: Devuelve la raíz del árbol binario pasado por parámetro.
Aliasing: El elemento se devuelve por referencia.

IZQ(**in** a: `ab`(α)) $\rightarrow$ *res* : α
Pre $\equiv$ {$\neg$ nil?(a)}
Post $\equiv$ {esAlias(*res*, izq(a))}
Complejidad: $\Theta(1)$
Descripción: Devuelve el subárbol izquierdo del árbol binario pasado por parámetro.
Aliasing: El subárbol se devuelve por referencia.

DER(**in** a: `ab`(α)) $\rightarrow$ *res* : α
Pre $\equiv$ {$\neg$ nil?(a)}
Post $\equiv$ {esAlias(*res*, der(a))}
Complejidad: $\Theta(1)$
Descripción: Devuelve el subárbol derecho del árbol binario pasado por parámetro.
Aliasing: El subárbol se devuelve por referencia.

ALTURA(**in** a: `ab(`α`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ altura(a)}
Complejidad: $\Theta(1)$
Descripción: Devuelve la máxima distancia entre la raíz del árbol binario y alguna de sus hojas.

CANTNODOS(**in** a: `ab(`α`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ tamaño(a)}
Complejidad: $\Theta(1)$
Descripción: Devuelve la cantidad de nodos del árbol binario.

INORDER(**in** a: `ab(`α`)`) $\rightarrow$ res : `lista(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ inorder(a)}
Complejidad: $\Theta(n)$
Descripción: Devuelve una lista con todos los elementos del árbol, recorridos en inorden.

PREORDER(**in** a: `ab(`α`)`) $\rightarrow$ res : `lista(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ preorder(a)}
Complejidad: $\Theta(n)$
Descripción: Devuelve una lista con todos los elementos del árbol, recorridos en preorden.

POSTORDER(**in** a: `ab(`α`)`) $\rightarrow$ res : `lista(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ postorder(a)}
Complejidad: $\Theta(n)$
Descripción: Devuelve una lista con todos los elementos del árbol, recorridos en postorden.

Representación

`ab(`α`)` **se representa con** `puntero(nodo)`

donde `nodo` es `tupla(`*valor*: α,
izq: `puntero(nodo)`,
der: `puntero(nodo)`)

Rep : puntero(nodo) $\longrightarrow$ bool

$$\begin{aligned}\text{Rep}(a) \equiv\ & \text{true} \iff \emptyset?(\text{padres}(a, \text{nodos}(a))) \wedge \\ & (\forall\, n : \text{nodo})\ (n \in \text{nodos}(a)\ \Rightarrow\ (\\ & ((\&n \neq a)\ \Rightarrow\ \#(\text{padres}(n, \text{nodos}(a))) = 1) \wedge \\ & n.izq \neq \text{NULL}\ \Rightarrow\ n.izq \neq n.der \wedge\end{aligned}$$

Abs : puntero(nodo) a $\longrightarrow$ ab(α) {Rep(a)}

$$\text{Abs}(a) \equiv \textbf{if}\ a = \text{NULL}\ \textbf{then}\ \text{nil}\ \textbf{else}\ \text{bin}(\text{Abs}(a \rightarrow izq), a \rightarrow valor, \text{Abs}(a \rightarrow der))\ \textbf{fi}$$

hijos : nodo $\longrightarrow$ conj(nodo)

$$\begin{aligned}\text{hijos}(n) \equiv\ & \textbf{if}\ n.izq = \text{NULL}\ \textbf{then}\ \emptyset\ \textbf{else}\ \text{Ag}(*(n.izq), \text{hijos}(*(n.izq)))\ \textbf{fi} \\ & \cup\ \textbf{if}\ n.der = \text{NULL}\ \textbf{then}\ \emptyset\ \textbf{else}\ \text{Ag}(*(n.der), \text{hijos}(*(n.der)))\ \textbf{fi}\end{aligned}$$

nodos : puntero(nodo) $\longrightarrow$ conj(nodo)

$$\text{nodos}(a) \equiv \textbf{if}\ a = \text{NULL}\ \textbf{then}\ \emptyset\ \textbf{else}\ \text{Ag}(*(a), \text{hijos}(*(a)))\ \textbf{fi}$$

altura : puntero(nodo) $\longrightarrow$ nat

altura(a) $\equiv$ **if** $a =$ NULL **then** 0 **else** 1 + máx(altura($a \rightarrow izq$), altura($a \rightarrow der$)) **fi**

padres : puntero(nodo) $\times$ conj(nodo) $\longrightarrow$ conj(nodo)

padres(a, ns) $\equiv$ **if** dameUno(ns).$izq = a \vee$ dameUno(ns).$der = a$ **then**
Ag(dameUno(ns), padres(a, sinUno(ns))
else
padres(a, sinUno(ns))
fi

Algoritmos

iNil() $\rightarrow$ *res* : `puntero(nodo)`

1 *res* $\leftarrow$ NULL ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iBin(**in** i: `puntero(nodo)`, **in** r: α, **in** d: `puntero(nodo)`) $\rightarrow$ *res* : `puntero(nodo)`

1 *res* $\leftarrow$ &$\langle r, i, d, 1 +$ max(Altura(i), Altura(d)), $1 +$ CantNodos(i) + CantNodos(d) $\rangle$ // Reservamos memoria para el nuevo nodo ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iEsNil(**in** a: `ab`(α)) $\rightarrow$ *res* : `bool`

1 *res* $\leftarrow a =$ NULL ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iRaiz(**in** a: `ab`(α)) $\rightarrow$ *res* : α

1 *res* $\leftarrow a \rightarrow valor$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iIzq(**in** a: `ab`(α)) $\rightarrow$ *res* : `ab`(α)

1 *res* $\leftarrow a \rightarrow izq$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iDer(**in** a: `ab`(α)) $\rightarrow$ *res* : `ab`(α)

1 *res* $\leftarrow a \rightarrow der$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

iAltura(**in** a: `ab`(α)) $\rightarrow$ *res* : `nat`

1 **if** EsNil(a) **then** ▷ $\Theta(1)$
2 | *res* $\leftarrow 0$ ▷ $\Theta(1)$
3 **else**
4 | *res* $\leftarrow 1 +$ max(Altura(Izq(a)), Altura(Der(a))) ▷ $\Theta(n)$ (ver justificación)
5 **end if**

Complejidad: $\Theta(n)$

Justificación: Cada nodo interior del árbol llama a la función recursivamente sobre sus dos hijos, por lo que la función se ejecuta exactamente una vez por cada uno de los n nodos del árbol. Como las operaciones que se realizan, sin contar la llamada recursiva, tienen complejidad $\Theta(1)$, la complejidad total resulta $\Theta(n)$.

---

iCantNodos(**in** a : ab(α)) $\rightarrow$ res : nat

```
1 if EsNil(a) then                                                    ▷ Θ(1)
2 |  res ← 0                                                         ▷ Θ(1)
3 else
4 |  res ← 1 + CantNodos(Izq(a)) + CantNodos(Der(a))                 ▷ Θ(n) (ver justificación)
5 end if
```

Complejidad: $\Theta(n)$

Justificación: Cada nodo interior del árbol llama a la función recursivamente sobre sus dos hijos, por lo que la función se ejecuta exactamente una vez por cada uno de los n nodos del árbol. Como las operaciones que se realizan, sin contar la llamada recursiva, tienen complejidad $\Theta(1)$, la complejidad total resulta $\Theta(n)$.

3. Módulo Diccionario Logarítmico

Notas preliminares

En todos los casos, al indicar las complejidades de los algoritmos, las variables que se utilizan corresponden a:

- n: Cantidad de claves definidas en el diccionario.

Servicios usados: puntero, tupla, nat

Interfaz

parámetros formales

géneros κ, σ

función $\bullet = \bullet(\textbf{in } k_1 : \kappa, \textbf{in } k_2 : \kappa) \to res$: `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} (k_1 = k_2)\}$
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ

función $\bullet \leq \bullet(\textbf{in } k_1 : \kappa, \textbf{in } k_2 : \kappa) \to res$: `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} (k_1 \leq k_2)\}$
Complejidad: $\Theta(order(k_1, k_2))$
Descripción: función de comparación por orden total estricto de κ

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

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

se explica con: DICCIONARIO(κ, σ)

géneros: `diccLog(`κ`, `σ`)`

Operaciones de diccionario

VACIO() $\to res$: `diccLog(`κ`, `σ`)`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{vacío}\}$
Complejidad: $\Theta(1)$
Descripción: Crea y devuelve un diccionario logarítmico vacío.

DEFINIR(**in/out** d: `diccLog(`κ`, `σ`)`, **in** $k : \kappa$, **in** $s : \sigma$)
Pre $\equiv \{d =_{\text{obs}} d_0\}$
Post $\equiv \{d =_{\text{obs}} \text{definir}(k, s, d_0)\}$
Complejidad: $\Theta(\log(n) \times order(k) + copy(k) + copy(s))$
Descripción: Define en el diccionario la clave pasada por parámetro con el significado pasado por parámetro. En caso de que la clave ya esté definida, sobreescribe su significado con el nuevo.
Aliasing: Las claves y significados se almacenan por copia.

BORRAR(**in/out** d: `diccLog(`κ`, `σ`)`, **in** $k : \kappa$)
Pre $\equiv \{d =_{\text{obs}} d_0 \wedge \text{def?}(k, d)\}$
Post $\equiv \{d =_{\text{obs}} \text{borrar}(k, d_0)\}$

Complejidad: $\Theta(\log(n) \times order(k))$
Descripción: Elimina del diccionario la clave pasada por parámetro.

CantClaves(**in** d: `diccLog(`κ`, `σ`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ #(claves(d))}
Complejidad: $\Theta(n)$
Descripción: Devuelve la cantidad de claves del diccionario.

Definido?(**in** d: `diccLog(`κ`, `σ`)`, **in** k: κ) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(k, d)}
Complejidad: $\Theta(\log(n) \times order(k))$
Descripción: Devuelve true si y solo si la clave pasada por parámetro está definida en el diccionario.

Obtener(**in** d: `diccLog(`κ`, `σ`)`, **in** k: κ) $\rightarrow$ res : σ
Pre $\equiv$ {def?(k, d)}
Post $\equiv$ {alias($res =_{\text{obs}}$ obtener(k, d))}
Complejidad: $\Theta(\log(n) \times order(k))$
Descripción: Devuelve el significado con el que la clave pasada por parámetro está definida en el diccionario.
Aliasing: El significado se pasa por referencia. Modificarlo implica cambiarlo en la estructura interna del diccionario.

Representación

`diccLog(`κ`, `σ`)` **se representa con** `estrAVL`

donde `estrAVL` es `ab(tupla(`*clave:* κ`, `*significado:* σ`, `*altSubarbol:* `nat`, `*padre:* `estrAVL))`

Rep : estrAVL $\longrightarrow$ bool

$$
\begin{aligned}
\text{Rep}(a) \equiv\ & \text{true} \iff \neg\ \text{nil?}(a) \Rightarrow\ (\\
& (\text{nil?}(\text{izq}(a)) \vee_{\text{L}} (\\
& \quad \text{raiz}(\text{izq}(a)).\mathit{clave} \neq \text{raiz}(a).\mathit{clave} \wedge \\
& \quad \text{raiz}(\text{izq}(a)).\mathit{clave} \leq \text{raiz}(a).\mathit{clave} \wedge \\
& \quad \text{raiz}(\text{izq}(a)).\mathit{padre} = a \\
&)) \wedge \\
& (\text{nil?}(\text{der}(a)) \vee_{\text{L}} (\\
& \quad \text{raiz}(\text{der}(a)).\mathit{clave} \neq \text{raiz}(a).\mathit{clave} \wedge \\
& \quad \text{raiz}(\text{der}(a)).\mathit{clave} \geq \text{raiz}(a).\mathit{clave} \wedge \\
& \quad \text{raiz}(\text{der}(a)).\mathit{padre} = a \\
&)) \wedge \\
& \text{raiz}(a).\mathit{altSubarbol} = \text{altura}(a) \wedge \\
& \text{raiz}(a).\text{altSubarbol} = \text{altura}(a) \wedge \\
& \text{máx}(\text{altura}(\text{izq}(a)), \text{altura}(\text{der}(a))) - \text{mín}(\text{altura}(\text{izq}(a)), \text{altura}(\text{der}(a))) \leq 1 \wedge \\
& \text{Rep}(\text{izq}(a)) \wedge \text{Rep}(\text{der}(a)) \\
&)
\end{aligned}
$$

Abs : estrAVL a $\longrightarrow$ dicc(κ, σ)) {Rep(a)}

Abs(a) $\equiv$ **if** nil?(a) **then** vacío **else** definir(raiz(a).clave, raiz(a).significado, unir(Abs(izq(a)), Abs(der(a)))) **fi**

unir : dicc($\kappa \times \sigma$) $\times$ dicc($\kappa \times \sigma$) $\longrightarrow$ dicc(κ, σ)

$$
\begin{array}{lll}
\text{unir}(d_1, d_2) & \equiv & \textbf{if } \text{vacío}(d_2) \textbf{ then} \\
& & \quad d_1 \\
& & \textbf{else} \\
& & \quad \text{definir}(\\
& & \quad\quad \text{dameUno}(\text{claves}(d_2)), \\
& & \quad\quad \text{obtener}(\text{dameUno}(\text{claves}(d_2)),\ d_2), \\
& & \quad\quad \text{unir}(d_1,\ \text{borrar}(\text{dameUno}(\text{claves}(d_2)),\ d_2)) \\
& & \quad) \\
& & \textbf{fi}
\end{array}
$$

Algoritmos

⚠ Algoritmo modificado

iVacio() → *res* : estrAVL

```
1 res ← Nil()                                                    ▷ Θ(1)
```

Complejidad: $\Theta(1)$

⚠ Algoritmo modificado

iDefinir(**in** k: κ, **in** s: σ, **in/out** a: estrAVL)

```
1  estrAVL padre
2  estrAVL lugar ← Buscar(a, k, padre)                           ▷ Θ(log(n) × order(k))
3  if ¬ EsNil(lugar) then                                        ▷ Θ(1)
4  |  Raiz(lugar).significado ← Copiar(s)                       ▷ Θ(copy(s))
5  else
6  |  estrAVL nuevo ← & Bin(Nil(), ⟨Copiar(k), Copiar(s), 1, padre⟩, Nil()) // Reservamos memoria para el
   |  nuevo nodo                                                ▷ Θ(copy(k) + copy(s))
7  |  if k ≤Raiz(padre).clave then                              ▷ Θ(order(k))
8  |  |  padre ← Bin(nuevo, Raiz(padre), Der(padre))            ▷ Θ(1)
9  |  else
10 |  |  padre ← Bin(Izq(padre), Raiz(padre), nuevo)            ▷ Θ(1)
11 |  end if
12 |  RebalancearArbol (padre)                                  ▷ Θ(log(n))
13 end if
```

Complejidad: $\Theta(\log(n) \times order(k) + copy(k) + copy(s))$

Justificación: La función tiene llamadas a funciones con complejidad $\Theta(\log(n) \times order(k))$ y $\Theta(copy(k) + copy(s))$.

⚠ Algoritmo modificado

iObtener(**in** a: estrAVL, **in** k: κ) → *res* : σ

```
1 estrAVL padre
2 estrAVL lugar ← Buscar(a, k, padre)                            ▷ Θ(log(n) × order(k))
3 res ← Raiz(lugar).significado                                  ▷ Θ(1)
```

Complejidad: $\Theta(\log(n) \times order(k))$

Justificación: La función tiene llamadas a funciones con complejidad $\Theta(\log(n) \times order(k)$ y $\Theta(copy(k) + copy(s))$.

 Algoritmo modificado

```
iCantClaves(in a: estrAVL) → res : nat
1 res ← CantNodos(a)                                   ▷ Θ(n)
```

Complejidad: $\Theta(n)$

 Algoritmo modificado

```
iDefinido?(in a: estrAVL, in k: κ) → res : bool
1 estrAVL padre
2 estrAVL lugar ← Buscar(a, k, padre)                  ▷ Θ(log(n) × order(k))
3 res ← ¬ EsNil(lugar)                                 ▷ Θ(1)
```

Complejidad: $\Theta(\log(n) \times order(k))$

Justificación: $\Theta(1) + \Theta(\log(n) \times order(k)) + \Theta(1) = \Theta(\log(n) \times order(k)) + \Theta(1)$

iBorrar(**in/out** a: estrAVL, **in** k: κ)

```
1  estrAVL padre
2  estrAVL lugar ← Buscar(a, k, padre)                          ▷ Θ(log(n) × order(k))
3  if EsNil(Izq(lugar)) ∧ EsNil(Der(lugar)) then                ▷ Θ(1)
4  |  if ¬ EsNil(padre) then                                    ▷ Θ(1)
5  |  |  if Izq(padre) = lugar then                            ▷ Θ(1)
6  |  |  |  padre ← Bin(Nil(), Raiz(padre), Der(padre))        ▷ Θ(1)
7  |  |  else
8  |  |  |  padre ← Bin(Izq(padre), Raiz(padre), Nil())       ▷ Θ(1)
9  |  |  end if
10 |  |  RebalancearArbol(padre)                              ▷ Θ(log(n))
11 |  else
12 |  |  a ← Nil()                                            ▷ Θ(1)
13 |  end if
14 else if EsNil(Der(lugar)) then                             ▷ Θ(1)
15 |  Raiz(Izq(lugar)).padre ← padre                          ▷ Θ(1)
16 |  if ¬ EsNil(padre) then                                  ▷ Θ(1)
17 |  |  if Izq(padre) = lugar then                           ▷ Θ(1)
18 |  |  |  padre ← Bin(Izq(lugar), Raiz(padre), Der(padre))  ▷ Θ(1)
19 |  |  else
20 |  |  |  padre ← Bin(Izq(padre), Raiz(padre), Izq(lugar))  ▷ Θ(1)
21 |  |  end if
22 |  |  RebalancearArbol(padre)                              ▷ Θ(log(n))
23 |  else
24 |  |  a ← Izq(lugar)                                       ▷ Θ(1)
25 |  end if
26 else if EsNil(Izq(lugar)) then                             ▷ Θ(1)
27 |  Raiz(Izq(lugar)).padre ← padre                          ▷ Θ(1)
28 |  if ¬ EsNil(padre) then                                  ▷ Θ(1)
29 |  |  if Izq(padre) = lugar then                           ▷ Θ(1)
30 |  |  |  padre ← Bin(Der(lugar), Raiz(padre), Der(padre))  ▷ Θ(1)
31 |  |  else
32 |  |  |  padre ← Bin(Izq(padre), Raiz(padre), Der(lugar))  ▷ Θ(1)
33 |  |  end if
34 |  |  RebalancearArbol(padre)                              ▷ Θ(log(n))
35 |  else
36 |  |  a ← Der(lugar)                                       ▷ Θ(1)
37 |  end if
```

iBorrar *(cont.)*

```
38 else
39 |  estrAVL reemplazo ← Der(lugar)                                   ▷ Θ(1)
40 |  if EsNil(Izq(reemplazo)) then                                    ▷ Θ(1)
41 |  |  if ¬ EsNil(padre) then                                        ▷ Θ(1)
42 |  |  |  if Izq(padre) = lugar then                                ▷ Θ(1)
43 |  |  |  |  Izq(padre) ← reemplazo                                 ▷ Θ(1)
44 |  |  |  else
45 |  |  |  |  Der(padre) ← reemplazo                                 ▷ Θ(1)
46 |  |  |  end if
47 |  |  else
48 |  |  |  a ← reemplazo                                             ▷ Θ(1)
49 |  |  end if
50 |  |  Raiz(reemplazo).padre ← padre                                ▷ Θ(1)
51 |  |  Izq(reemplazo) ← Izq(lugar)                                  ▷ Θ(1)
52 |  |  Izq(lugar).padre ← reemplazo                                 ▷ Θ(1)
53 |  |  RebalancearArbol(reemplazo)                                  ▷ Θ(log(n))
54 |  else
55 |  |  while ¬ EsNil(Izq(reemplazo)) do                             ▷ Θ(log(n)) iteraciones
56 |  |  |  reemplazo ← Izq(reemplazo)                                ▷ Θ(1)
57 |  |  end while
58 |  |  estrAVL padreReemplazo ← Raiz(reemplazo).padre               ▷ Θ(1)
59 |  |  if ¬ EsNil(padre) then                                       ▷ Θ(1)
60 |  |  |  if Izq(padre) = lugar then                                ▷ Θ(1)
61 |  |  |  |  padre ← Bin(reemplazo, Raiz(padre), Der(padre))
62 |  |  |  else
63 |  |  |  |  padre ← Bin(Izq(padre), Raiz(padre), reemplazo)
64 |  |  |  end if
65 |  |  else
66 |  |  |  a.raiz ← reemplazo                                        ▷ Θ(1)
67 |  |  end if
68 |  |  (reemplazo → padre) ← padre                                  ▷ Θ(1)
69 |  |  (reemplazo → izq) ← lugar → izq                              ▷ Θ(1)
70 |  |  (lugar → izq → padre) ← reemplazo                            ▷ Θ(1)
71 |  |  (padreReemplazo → izq) ← (reemplazo → der)                   ▷ Θ(1)
72 |  |  if (reemplazo → der) ≠ NULL then                             ▷ Θ(1)
73 |  |  |  (reemplazo → der → padre) ← padreReemplazo                ▷ Θ(1)
74 |  |  end if
75 |  |  (reemplazo → der) ← (lugar → der)                            ▷ Θ(1)
76 |  |  (lugar → der → padre) ← reemplazo                            ▷ Θ(1)
77 |  |  RebalancearArbol(reemplazo)                                  ▷ Θ(log(n))
78 |  end if
79 end if
80 delete(lugar) // Liberamos la memoria ocupada por el nodo eliminado.  ▷ Θ(1)
```

Complejidad: $\Theta(\log(n) \times order(k))$

Justificación: El algoritmo tiene una llamada a función con complejidad $\Theta(\log(n) \times order(k))$, y luego presenta varios casos, pero en todos ellos las funciones llamadas son $O(\log(n))$.

Algoritmo modificado

```
iBuscar(in a: estrAVL, in k: κ, out padre: estrAVL) → res : puntero(estrAVL)
1  padre ← Nil()                                                          ▷ Θ(1)
2  actual ← a                                                             ▷ Θ(1)
3  while ¬ EsNil(actual) ∧_L (Raiz(actual).clave ≠ k) do                  ▷ Θ(log(n)) iteraciones
4  |  padre ← actual                                                      ▷ Θ(1)
5  |  if k ≤ Raiz(actual).clave then                                      ▷ Θ(order(k))
6  |  |  actual ← Izq(actual)                                            ▷ Θ(1)
7  |  else
8  |  |  actual ← Der(actual)                                            ▷ Θ(1)
9  |  end if
10 end while
11 res ← actual                                                           ▷ Θ(1)
```

Descripción: Esta operación privada recibe el árbol AVL sobre el que está representado el diccionario y una clave por parámetro. Si la clave está definida, devuelve el subárbol que tiene a dicha clave en su raíz, y coloca en el parámetro de out *padre* el subárbol del cual dicha clave es hija inmediata. En caso contrario, devuelve un árbol vacío y coloca en el parámetro de out *padre* el subárbol árbol del cual la clave debería ser hija inmediata, si estuviera definida.

Complejidad: $\Theta(\log(n) \times order(k))$

Justificación: El algoritmo presenta un ciclo que se repite $\Theta(\log(n))$ veces, y en cada una de ellas se realiza una llamada a función con complejidad $\Theta(order(k))$.

Algoritmo modificado

```
iRecalcularAltura(in a: estrAVL)
1  if ¬ EsNil(Izq(a)) ∧ ¬ EsNil(Izq(a)) then                              ▷ Θ(1)
2  |  Raiz(a).altSubarbol ← 1 + max(Raiz(Izq(a)).altSubarbol, Raiz(Der(a)).altSubarbol)   ▷ Θ(1)
3  else if ¬ EsNil(Izq(a)) then                                           ▷ Θ(1)
4  |  Raiz(a).altSubarbol ← 1 + Raiz(Izq(a)).altSubarbol                  ▷ Θ(1)
5  else if ¬ EsNil(Der(a)) then                                           ▷ Θ(1)
6  |  Raiz(a).altSubarbol ← 1 + Raiz(Der(a)).altSubarbol                  ▷ Θ(1)
7  else
8  |  Raiz(a).altSubarbol ← 1                                            ▷ Θ(1)
9  end if
```

Descripción: Esta operación privada recibe un subárbol y recalcula el valor de su campo *altSubarbol* en función a los datos que sus nodos hijos poseen en este campo.

Complejidad: $\Theta(1)$

Justificación: El algoritmo presenta varios casos, y todos ellos realizan realizan operaciones con complejidad $\Theta(1)$.

Algoritmo modificado

```
iFactorDeBalanceo(in a: estrAVL) → res : int
1  int altIzq ← EsNil(Izq(a)) ? 0 : Raiz(Izq(a)).altSubarbol              ▷ Θ(1)
2  int altDer ← EsNil(Der(a)) ? 0 : Raiz(Der(a)).altSubarbol              ▷ Θ(1)
3  res ← altDer − altIzq                                                  ▷ Θ(1)
```

Descripción: Esta operación privada recibe un subárbol cualquiera y calcula su factor de balanceo.

Complejidad: $\Theta(1)$

Justificación: $\Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$

Algoritmo modificado

```
iRotarAIzquierda(in/out a : estrAVL)
1  estrAVL nuevo_1 ← Bin(Izq(a), Raiz(a), Izq(Der(a)))                          ▷ Θ(1)
2  estrAVL nuevo_2 ← Bin(nuevo_1, Raiz(Der(a)), Der(Der(a)))                    ▷ Θ(1)
3  delete(Der(a)) // Liberamos la memoria de los subárboles que ya no usaremos  ▷ Θ(1)
4  delete(a)                                                                    ▷ Θ(1)
5  a ← nuevo_2                                                                  ▷ Θ(1)
6  Raiz(a).padre ← Raiz(Izq(a)).padre                                           ▷ Θ(1)
7  Raiz(Izq(a)).padre ← a                                                       ▷ Θ(1)
8  Raiz(Der(a)).padre ← a                                                       ▷ Θ(1)
9  Raiz(Izq(Izq(a))).padre ← Izq(a)                                             ▷ Θ(1)
10 Raiz(Der(Izq(a))).padre ← Izq(a)                                             ▷ Θ(1)
11 RecalcularAltura(Izq(a))                                                     ▷ Θ(1)
12 RecalcularAltura(a)                                                          ▷ Θ(1)
```

Descripción: Esta operación privada recibe un subárbol y realiza una rotación a izquierda de su raíz. ¡Ojo, rompe el invariante de representación! (Los campos *altSubarbol* de los subárboles superiores quedan inconsistentes).

Complejidad: $\Theta(1)$

Justificación: Todas las operaciones que realiza el algoritmo tienen complejidad $\Theta(1)$.

Algoritmo modificado

```
iRotarADerecha(in/out a : estrAVL)
1  estrAVL nuevo_1 ← Bin(Der(Izq(a)), Raiz(a), Der(a))                          ▷ Θ(1)
2  estrAVL nuevo_2 ← Bin(Izq(Izq(a)), Raiz(Izq(a)), nuevo_1)                    ▷ Θ(1)
3  delete(Izq(a)) // Liberamos la memoria de los subárboles que ya no usaremos  ▷ Θ(1)
4  delete(a)                                                                    ▷ Θ(1)
5  a ← nuevo_2                                                                  ▷ Θ(1)
6  Raiz(a).padre ← Raiz(Der(a)).padre                                           ▷ Θ(1)
7  Raiz(Izq(a)).padre ← a                                                       ▷ Θ(1)
8  Raiz(Der(a)).padre ← a                                                       ▷ Θ(1)
9  Raiz(Izq(Der(a))).padre ← Der(a)                                             ▷ Θ(1)
10 Raiz(Der(Der(a))).padre ← Der(a)                                             ▷ Θ(1)
11 RecalcularAltura(Der(a))                                                     ▷ Θ(1)
12 RecalcularAltura(a)                                                          ▷ Θ(1)
```

Descripción: Esta operación privada recibe un subárbol y realiza una rotación a derecha de su raíz. ¡Ojo, rompe el invariante de representación! (Los campos *altSubarbol* de los subárboles superiores quedan inconsistentes).

Complejidad: $\Theta(1)$

Justificación: Todas las operaciones que realiza el algoritmo tienen complejidad $\Theta(1)$.

Algoritmo modificado

```
iRebalancearArbol(in a : estrAVL)
1  estrAVL p ← a                                                          ▷ Θ(1)
2  while ¬ EsNil(p) do                                                    ▷ Θ(log(n)) iteraciones
3      RecalcularAltura(p)                                                ▷ Θ(1)
4      int fdb1 ← FactorDeBalanceo(p)                                     ▷ Θ(1)
5      if fdb1 = 2 then                                                   ▷ Θ(1)
6          estrAVL q ← Der(p)                                             ▷ Θ(1)
7          int fdb2 ← FactorDeBalanceo(q)                                 ▷ Θ(1)
8          if fdb2 = 1 ∨ fdb2 = 0 then                                    ▷ Θ(1)
9              RotarAIzquierda(p)                                         ▷ Θ(1)
10         else if fdb2 = −1 then                                         ▷ Θ(1)
11             RotarADerecha(q)                                           ▷ Θ(1)
12             RotarAIzquierda(p)                                         ▷ Θ(1)
13         end if
14     else if fdb1 = −2 then                                             ▷ Θ(1)
15         estrAVL q ← Izq(p)                                             ▷ Θ(1)
16         int fdb2 ← FactorDeBalanceo(q)                                 ▷ Θ(1)
17         if fdb2 = −1 ∨ fdb2 = 0 then                                   ▷ Θ(1)
18             RotarADerecha(p)                                           ▷ Θ(1)
19         else if fdb2 = 1 then                                          ▷ Θ(1)
20             RotarAIzquierda(q)                                         ▷ Θ(1)
21             RotarADerecha(p)                                           ▷ Θ(1)
22         end if
23     end if
24     p ← Raiz(p).padre                                                  ▷ Θ(1)
25 end while
```

Descripción: Esta operación privada recibe un subárbol y, utilizando la información del campo *padre* de su raíz, restaura el invariante de representación en la rama ascendente a partir de ella, realizando las rotaciones necesarias para rebalancear el árbol.

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

Justificación: El algoritmo presenta un ciclo que se ejecuta $\Theta(\log(n))$ veces, y en cada una de ellas se realizan operaciones con complejidad $\Theta(1)$.