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 $\equiv$ $\{r =_{\text{obs}} r_0 \wedge (\forall c'\text{: compu})(c' \in \text{computadoras}(r) \rightarrow \text{ip}(c) \neq \text{ip}(c'))\}$
Post $\equiv$ $\{r =_{\text{obs}} \text{agregarCompu}(r_0, \text{c})\}$
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 $\equiv$ $\{r =_{\text{obs}} r_0 \wedge c_1 \in \text{computadoras}(r) \wedge c_2 \in \text{computadoras}(r) \wedge \text{ip}(c_0) \neq \text{ip}(c_1) \wedge \neg\text{conectadas?}(r, c_0, c_1) \wedge \neg\text{usaInterfaz?}(r, c_0, i_0) \wedge \neg\text{usaInterfaz?}(r, c_1, i_1)\}$
Post $\equiv$ $\{r =_{\text{obs}} \text{conectar}(r_0, c_0, i_0, c_1, i_1)\}$
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 $\equiv$ $\{\text{esAlias}(res, \text{computadoras(r)})\}$
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 $\equiv$ $\{c_0 \in \text{computadoras}(r) \wedge c_1 \in \text{computadoras}(r)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{conectadas?}(r, c_0, c_1)\}$
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 $\equiv$ $\{c_0 \in \text{computadoras}(r) \wedge c_1 \in \text{computadoras}(r) \wedge_{\text{L}} \text{conectadas?}(r, c_0, c_1)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{interfazUsada}(r, c_0, c_1)\}$
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 $\equiv$ $\{c \in \text{computadoras}(r)\}$

Post $\equiv \{res =_{\text{obs}}$ 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 $\equiv \{c \in$ computadoras(r)$\}$
Post $\equiv \{res =_{\text{obs}}$ 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 $\equiv \{c_0 \in$ computadoras(r) $\wedge$ $c_1 \in$ computadoras(r)$\}$
Post $\equiv \{res =_{\text{obs}}$ 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 $\equiv \{c_0 \in$ computadoras(r) $\wedge$ $c_1 \in$ computadoras(r)$\}$
Post $\equiv \{res =_{\text{obs}}$ 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 $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \#$(computadoras(r))$\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve cuántas computadoras hay en la red.

copiar(**in** r : `Red`) $\rightarrow res$: `Red`
Pre $\equiv$ {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 : Red $\longrightarrow$ bool

Rep(e) $\equiv$ ($\forall$c: compu)(c $\in$ ArmarComputadoras(e.compus) $\Rightarrow_{\text{L}}$ $\neg$ Pertenece?(e.compus, c, c))$\wedge$
#ArmarComputadoras(e.compus) = e.cantidadCompus $\wedge$
($\forall c_1$: compu)(($\forall c_2$: compu) ($c_1 \in$ ArmarComputadoras(e.compus) $\wedge$ $c_2 \in$ ArmarComputadoras(e.compus)
$\Rightarrow_{\text{L}}$ Pertenece?(e.compus, c_1, c_2) $\Leftrightarrow$ Pertenece?(e.compus, c_2, c_1))) $\wedge$
($\forall c_1$: compu)($c_1 \in$ ArmarComputadoras(e.compus) $\Rightarrow_{\text{L}}$ ($\forall c_2$: compu) (Pertenece?(e.compus, c_1, c_2) $\Rightarrow$ c_2
$\in$ ArmarComputadoras(e.compus))) $\wedge$
sinRepetidos(ArmarSecuencia(e.compus))

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($\langle\Pi_1$(prim(l)), GenerarInterfaces(Π_2(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**
Π_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 $\times$ secu(tupla(Interfaz $\times$ ItRed)))) l $\times$ compu c_1 $\times$ 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 $\times$ ItRed)) l $\times$ compu c $\longrightarrow$ Interfaz

$$
\text{DevolverInterfaz}(l, c) \equiv \begin{array}{l}
\textbf{if } (\Pi_1(c_2) = \Pi_1(\text{siguiente}(\Pi_2(\text{prim}(l))))) \textbf{ then} \\
\quad \Pi_1(\text{prim}(l)) \\
\textbf{else} \\
\quad \text{DevolverInterfaz}_{\text{aux}}(\text{fin}(l, \text{c})) \\
\textbf{fi}
\end{array}
$$

Algoritmos

Algoritmo renovado

iIniciarRed() $\rightarrow$ *res* : `estrRed`

$res \leftarrow \langle$Vacio(), Vacio()$\rangle$ $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

Algoritmo renovado

iAgregarCompu(**in/out** r : `estrRed`, **in** c : `compu`)

AgregarRapido($r.compus$, c) $\triangleright\ \Theta(1)$
DefinirRapido($r.conexiones$, $c.IP$, Vacio()) $\triangleright\ \Theta(1)$

Complejidad: $\Theta(1)$

Algoritmo renovado

iConectar(**in/out** r : `estrRed`), **in** c_1 : `compu`, **in** i_1 : `interfaz`, **in** c_2 : `compu`, **in** i_2 : `interfaz`)

`itDicc(IP, diccConexiones)` $it_1 \leftarrow$ CrearIt($r.conexiones$) $\triangleright\ \Theta(1)$
`itDicc(IP, diccConexiones)` $it_2 \leftarrow$ CrearIt($r.conexiones$) $\triangleright\ \Theta(1)$
while SiguienteClave(it_1) $\neq c_1.IP$ **do** $\triangleright\ \Theta(n)$ iteraciones
  Avanzar(it_1) $\triangleright\ \Theta(1)$
end while
while SiguienteClave(it_2) $\neq c_1.IP$ **do** $\triangleright\ \Theta(n)$ iteraciones
  Avanzar(it_2) $\triangleright\ \Theta(1)$
end while
DefinirRapido(SiguienteSignificado(it_1), i_1, Copiar(it_2)) $\triangleright\ \Theta(1)$
DefinirRapido(SiguienteSignificado(it_2), i_2, Copiar(it_1)) $\triangleright\ \Theta(1)$

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 renovado

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

`itDicc(IP, diccConexiones)` $it_1 \leftarrow$ CrearIt($r.conexiones$) $\triangleright\ \Theta(1)$
while SiguienteClave(it_1) $\neq c_1.IP$ **do** $\triangleright\ \Theta(n)$ iteraciones
    Avanzar(it_1) $\triangleright\ \Theta(1)$
end while
`itDicc(interfaz, itDicc(IP, diccConexiones))` $it_2 \leftarrow$ CrearIt(Significado(Siguiente(it_1))) $\triangleright\ \Theta(1)$
while HaySiguiente?(it_2) $\wedge_L$ SiguienteClave(SiguienteSignificado(it_2)) $\neq c_2.IP$ **do** $\triangleright\ \Theta(I)$ iteraciones
    Avanzar(it_2) $\triangleright\ \Theta(1)$
end while
$res \leftarrow$ HaySiguiente?(it_2) $\wedge_L$ SiguienteClave(SiguienteSignificado(it_2)) $= c_2.IP$ $\triangleright\ \Theta(1)$

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 renovado

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

`itLista(estrCompu)` $it_1 \leftarrow$ crearIt(r.compus) $\triangleright\ \Theta(1)$
while siguiente(it_1).IP $\neq c_1$.IP **do** $\triangleright\ \Theta(n)$ iteraciones
    avanzar(it_1) $\triangleright\ \Theta(1)$
end while
`itLista(tupla(interfaz, itLista(estrCompu)))` $it_2 \leftarrow$ crearIt(siguiente(it_1).conexiones) $\triangleright\ \Theta(1)$
while (siguiente(siguiente(it_2).com)).IP $\neq c_2$.IP **do** $\triangleright\ \Theta(I)$ iteraciones
    avanzar(it_1) $\triangleright\ \Theta(1)$
end while
$res \leftarrow$ siguiente(it_2).inter $\triangleright\ \Theta(1)$

Complejidad: $\Theta(n + I)$

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

$res \leftarrow$ vacio() $\triangleright\ \Theta(1)$
`itLista(estrComp)` $it_1 \leftarrow$ crearIt(r.compus) $\triangleright\ \Theta(1)$
while siguiente(it_1).IP $\neq c$.IP **do** $\triangleright\ \Theta(n)$ iteraciones
    avanzar(it_1) $\triangleright\ \Theta(1)$
end while
`itLista(tupla(interfaz, itLista(estrCompu)))` $it_2 \leftarrow$ crearIt(siguiente(it_1).conexiones) $\triangleright\ \Theta(1)$
while haySiguiente?(it_2) **do** $\triangleright\ \Theta(n)$ iteraciones
    if haySiguiente?(siguiente(it_2).com) **then** $\triangleright\ \Theta(1)$
        agregar(res, $\langle$siguiente(siguiente(it_2).com).IP,
        crearConjunto(siguiente(siguiente(it_2).com).conexiones))$\rangle$) $\triangleright\ \Theta(I^2)$
    end if
    avanzar(it_2) $\triangleright\ \Theta(1)$
end while

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

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

`nat` $n \leftarrow 0$ ▷ $\Theta(1)$
$res \leftarrow$ vacio() ▷ $\Theta(1)$
while n <longitud(l) **do** ▷ $\Theta(I)$ iteraciones
  agregar(res, (l[n]).inter) ▷ $\Theta(I)$
  $n \leftarrow n + 1$ ▷ $\Theta(1)$
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`) $\rightarrow$ res : `bool`

`itLista(estrComp)` $it_1 \leftarrow$ crearIt(r.compus) ▷ $\Theta(1)$
while siguiente(it_1).IP $\neq c$.IP **do** ▷ $\Theta(n)$ iteraciones
  avanzar(it_1) ▷ $\Theta(1)$
end while
`itLista(tupla(interfaz, itLista(estrCompu)))` $it_2 \leftarrow$ crearIt(siguiente(it_1).conexiones) ▷ $\Theta(1)$
while siguiente(it_2).inter $\neq i$ **do** ▷ $\Theta(I)$ iteraciones
  avanzar(it_2) ▷ $\Theta(1)$
end while
$res \leftarrow$ haySiguiente(siguiente(it_2).com) ▷ $\Theta(1)$

Complejidad: $\Theta(n + I)$

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

$res \leftarrow$ vacio() ▷ $\Theta(1)$
if pertenece?(c_2, vecinos(r, c_1)) **then** ▷ $\Theta(I)$
  agregar(res, agregarAtras(agregarAtras(<>,c_1), c_2)) ▷ $\Theta(n + I)$
else
  $res \leftarrow$ dameMinimos(Caminos(r, c_1, c_2, agregarAtras(<>,c_1), pasarConjASecu(vecinos(r, c_1)))) ▷ $\Theta(n^3 \times n! \times n!)$
end if

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

dameMinimos(**in** c: `conj(lista(compu))`) $\rightarrow$ res : `conj(lista(compu))`

if esVacio?(c) **then** ▷ $\Theta(1)$
  $res \leftarrow$ vacio() ▷ $\Theta(1)$
else
  `itConj(lista(compu))` $it \leftarrow$ crearIt(c) ▷ $\Theta(1)$
  $res \leftarrow$ dameMinimosAux(c, minimaLong(c, long(siguiente(it)))) ▷ $\Theta(n \times n!)$
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) $\rightarrow$ res : conj(lista(compu))

```
itConj(lista(compu)) it ← crearIt(c)                ▷ Θ(1)
res ← vacio()                                       ▷ Θ(1)
while haySiguiente(it) do                           ▷ Θ(n!) iteraciones
    if long(siguiente(it))= n then                  ▷ Θ(1)
        agregar(res, siguiente(it))                 ▷ Θ(n)
        avanzar(it)                                 ▷ Θ(1)
    else
        avanzar(it)                                 ▷ Θ(1)
    end if
end while
```

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

minimaLong(**in** c: conj(lista(compu)), **in** n: nat) $\rightarrow$ res : nat

```
nat i ← n                                           ▷ Θ(1)
itConj(lista(compu)) it ← crearIt(c)                ▷ Θ(1)
while haySiguiente(it) do
    if long(siguiente(it)) then
        i ← longitud(siguiente(it))                 ▷ Θ(1)
        avanzar(it)                                 ▷ Θ(1)
    else
        avanzar(it)                                 ▷ Θ(1)
    end if
end while
res ← i                                             ▷ Θ(1)
```

Complejidad: $\Theta(n!)$

Justificación: Devuelve la longitud de la secuencia más chica

pasarConjASecu(**in** c: conj(compu)) $\rightarrow$ res : secu(compu)

```
res ← vacia()                                       ▷ Θ(1)
ItConj it ← crearIt(c)                              ▷ Θ(1)
while haySiguiente(it) do                           ▷ Θ(n) iteraciones
    agregarAtras(res, siguiente(it))                ▷ Θ(I)
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** c_1 : compu, **in** c_2 : compu) $\rightarrow$ res : bool

```
res ← (¬esVacio?(iCaminosMinimos(r, c1, c2)))      ▷ Θ(n^2 × n!)
```

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

iCaminos(**in** r: estrRed, **in** c_1 : compu, **in** c_2 : compu, **in** l: lista(estrCompu), **in** vec: lista(estrCompu)) $\rightarrow$ res : conj(lista(estrCompu))

```
if vacia?(vec) then
    res ← vacia()
else
    if último(l) = c1 then
        res ← agregar(l, vacia())
    else
        if ¬está?(primero(vec, l)) then
            res ← unión(caminos(r, c0, c1, agregarAtras(l, primero(vec)), Vecinos(r, primeros(vec))),
            caminos(r, c0, c1, l, fin(vec)))
        else
            res ← caminos(r, c0, c1, l, fin(vec))
        end if
    end if
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** c_1 : conj(lista(compu)), **in** c_2 : conj(lista(compu))) $\rightarrow$ res : conj(lista(compu))

```
res ← vacio()                                  ▷ Θ(1)
if vacio?(c1) then                             ▷ Θ(1)
    res ← c2                                   ▷ Θ(I × n × n!)
else
    itConj(lista(compu)) it ← crearIt(c1)      ▷ Θ(1)
    while haySiguiente(it) do                  ▷ Θ(n)
        Ag(siguiente(it), c2)                  ▷ Θ(n)
        avanzar(it)                            ▷ Θ(1)
    end while
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)) $\rightarrow$ res : bool

```
if vacia?(l) then                                    ▷ Θ(1)
    res ← false                                      ▷ Θ(1)
else
    ItLista(compu) it ← crearIt(l)                   ▷ Θ(1)
    while haySiguiente(it) ∧L siguiente(it) ≠ c do   ▷ Θ(n)

        avanzar(it)                                  ▷ Θ(1)
    end while
end if
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`) $\to$ res : `conj(compu)`

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

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

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

```
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`) $\to$ res : `nat`

```
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: ARBOL BINARIO(α)

géneros: `ab`(α)

Operaciones del TAD Árbol binario

NIL() $\rightarrow$ *res* : `ab`(α)
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{nil}\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{bin}(i, r, d) \land \text{esAlias}(\text{izq}(res), i) \land \text{esAlias}(\text{raiz}(res), r) \land \text{esAlias}(\text{der}(res), d)\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{nil?}(a)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve *true* si y solo si el árbol binario está vacío.

RAIZ(**in** a : `ab`(α)) $\rightarrow$ *res* : α
Pre $\equiv$ $\{\neg\ \text{nil?}(a)\}$
Post $\equiv$ $\{\text{esAlias}(res, \text{raiz}(a))\}$
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 $\equiv$ $\{\neg\ \text{nil?}(a)\}$
Post $\equiv$ $\{\text{esAlias}(res, \text{izq}(a))\}$
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 $\equiv$ $\{\neg\ \text{nil?}(a)\}$
Post $\equiv$ $\{\text{esAlias}(res, \text{der}(a))\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{altura}(a)\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{tamaño}(a)\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{inorder}(a)\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{preorder}(a)\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{postorder}(a)\}$
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

Rep(a) $\equiv$ true $\Longleftrightarrow$ $\emptyset$?(padres(a, nodos(a))) $\wedge$
($\forall$ n : nodo) ($n \in$ nodos(a) $\Rightarrow$ (
((&$n \neq a$) $\Rightarrow$ #(padres(n, nodos(a))) = 1) $\wedge$
$n.izq \neq$ NULL $\Rightarrow$ $n.izq \neq n.der$ $\wedge$

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

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

hijos : nodo $\longrightarrow$ conj(nodo)

hijos(n) $\equiv$ **if** $n.izq$ = NULL **then** $\emptyset$ **else** Ag($*(n.izq)$, hijos($*(n.izq)$)) **fi**
$\cup$ **if** $n.der$ = NULL **then** $\emptyset$ **else** Ag($*(n.der)$, hijos($*(n.der)$)) **fi**

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

nodos(a) $\equiv$ **if** a = NULL **then** $\emptyset$ **else** Ag($*(a)$, hijos($*(a)$)) **fi** altura : puntero(nodo) $\longrightarrow$ nat

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

NIL() $\to$ res : `puntero(nodo)`

1 $res \leftarrow$ NULL ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

BIN(**in** i: `puntero(nodo)`, **in** r: α, **in** d: `puntero(nodo)`) $\to$ 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: $\Theta(1)$

ESNIL(**in** a: `ab(α)`) $\to$ res : `bool`

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

Complejidad: $\Theta(1)$

RAIZ(**in** a: `ab(α)`) $\to$ res : α

1 $res \leftarrow a \to valor$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

IZQ(**in** a: `ab(α)`) $\to$ res : `ab(α)`

1 $res \leftarrow a \to izq$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

DER(**in** a: `ab(α)`) $\to$ res : `ab(α)`

1 $res \leftarrow a \to der$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

ALTURA(**in** a: `ab(α)`) $\to$ res : `nat`

```
if EsNil(a) then                                   ▷ Θ(1)
|  res ← 0                                         ▷ Θ(1)
else
|  res ← 1 + max(Altura(Izq(a)), Altura(Der(a)))   ▷ Θ(n) (ver justificación)
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 $\Theta(n)$.

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

```
if EsNil(a) then                                         ▷ Θ(1)
  | res ← 0                                              ▷ Θ(1)
else
  | res ← 1 + CantNodos(Izq(a)) + CantNodos(Der(a))      ▷ Θ(n) (ver justificación)
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 $\bullet = \bullet$(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} (k_1 = k_2)\}$
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ

función $\bullet \leq \bullet$(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {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** $k : \kappa$) $\rightarrow$ *res* : κ
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} k\}$
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ

función Copiar(**in** $s : \sigma$) $\rightarrow$ *res* : σ
Pre $\equiv$ {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() $\rightarrow$ *res* : `diccLog(`κ`,` σ`)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}}$ 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 $\equiv$ $\{d =_{\text{obs}}$ 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 $\equiv$ $\{d =_{\text{obs}} d_0 \wedge$ def?$(k, d)\}$
Post $\equiv$ $\{d =_{\text{obs}}$ 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 $\equiv$ $\{res =_{\text{obs}} \#(\text{claves}(d))\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve la cantidad de claves del diccionario.

Definido?(**in** d: `diccLog`(κ, σ), **in** k: κ) $\rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{def?}(k, d)\}$
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 $\equiv$ $\{\text{def?}(k, d)\}$
Post $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{obtener}(k, d))\}$
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`))`

Rep : estrAVL $\longrightarrow$ bool

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

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)

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

Algoritmos

iVacio() $\rightarrow res$: `estrAVL`

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

Complejidad: $\Theta(1)$

---

iDefinir(**in** $k: \kappa$, **in** $s: \sigma$, **in/out** a: estrAVL)

```
estrAVL padre
estrAVL lugar ← Buscar(a, k, padre)                                   ▷ Θ(log(n) × order(k))
if ¬ EsNil(lugar) then                                                ▷ Θ(1)
  | Raiz(lugar).signficado ← Copiar(s)                                ▷ Θ(copy(s))
else
  | estrAVL nuevo ← & Bin(Nil(), ⟨Copiar(k), Copiar(s)⟩, Nil()) // Reservamos memoria para el nuevo nodo
  |   ▷ Θ(copy(k) + copy(s))
  | if k ≤Raiz(padre).clave then                                      ▷ Θ(order(k))
  |   | padre ← Bin(nuevo, Raiz(padre), Der(padre))                   ▷ Θ(1)
  | else
  |   | padre ← Bin(Izq(padre), Raiz(padre), nuevo)                   ▷ Θ(1)
  | end if
  | RebalancearArbol (padre)                                          ▷ Θ(log(n))
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 $\Theta(copy(k) + copy(s))$.

---

iObtener(**in** a: estrAVL, **in** $k: \kappa$) $\rightarrow res : \sigma$

```
puntero(nodo) padre ← NULL                                            ▷ Θ(1)
puntero(nodo) lugar ← Buscar(a, k, padre)                             ▷ Θ(log(n) × order(k))
res ← (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 $\Theta(copy(k) + copy(s))$.

---

iCantClaves(**in** a: estrAVL) $\rightarrow res$: nat

```
res ← CantNodos(a)                                                    ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

iDefinido?(**in** a: estrAVL, **in** $k: \kappa$) $\rightarrow res$: bool

```
puntero(nodo) padre ← NULL                                            ▷ Θ(1)
puntero(nodo) lugar ← Buscar(a, k, padre)                             ▷ Θ(log(n) × order(k))
res ← (lugar ≠ NULL)                                                  ▷ 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)

```
puntero(nodo) padre ← NULL                                        ▷ Θ(1)
puntero(nodo) lugar ← Buscar(a, k, padre)                         ▷ Θ(log(n) × order(k))
if lugar → izq = NULL ∧ lugar → der = NULL then                   ▷ Θ(1)
    if padre ≠ NULL then                                          ▷ Θ(1)
        if padre → izq = lugar then                               ▷ Θ(1)
            (padre → izq) ← NULL                                  ▷ Θ(1)
        else
            (padre → der) ← NULL                                  ▷ Θ(1)
        end if
        RebalancearArbol(padre)                                   ▷ Θ(log(n))
    else
        a.raiz = NULL                                             ▷ Θ(1)
    end if
else if lugar → der = NULL then                                   ▷ Θ(1)
    (lugar → izq → padre) ← padre                                 ▷ Θ(1)
    if padre ≠ NULL then                                          ▷ Θ(1)
        if padre → izq = lugar then                               ▷ Θ(1)
            (padre → izq) ← (lugar → izq)                         ▷ Θ(1)
        else
            (padre → der) ← (lugar → izq)                         ▷ Θ(1)
        end if
        RebalancearArbol(padre)                                   ▷ Θ(log(n))
    else
        a.raiz ← lugar → izq                                      ▷ Θ(1)
    end if
else if lugar → izq = NULL then                                   ▷ Θ(1)
    (lugar → der → padre) ← padre                                 ▷ Θ(1)
    if padre ≠ NULL then                                          ▷ Θ(1)
        if padre → izq = lugar then                               ▷ Θ(1)
            (padre → izq) ← (lugar → der)                         ▷ Θ(1)
        else
            (padre → der) ← (lugar → der)                         ▷ Θ(1)
        end if
        RebalancearArbol(padre)                                   ▷ Θ(log(n))
    else
        a.raiz ← lugar → izq                                      ▷ Θ(1)
    end if
```

iBorrar *(cont.)*

```
else
    puntero(nodo) reemplazo ← (lugar → der)                          ▷ Θ(1)
    if (reemplazo → izq = NULL) then                                 ▷ Θ(1)
        if padre ≠ NULL then                                         ▷ Θ(1)
            if (padre → izq) = lugar then                            ▷ Θ(1)
                (padre → izq) ← reemplazo                            ▷ Θ(1)
            else
                (padre → der) ← reemplazo                            ▷ Θ(1)
            end if
        else
            a.raiz ← reemplazo                                       ▷ Θ(1)
        end if
        (reemplazo → padre) ← padre                                  ▷ Θ(1)
        (reemplazo → izq) ← lugar → izq                              ▷ Θ(1)
        (lugar → izq → padre) ← reemplazo                            ▷ Θ(1)
        RebalancearArbol(reemplazo)                                  ▷ Θ(log(n))
    else
        while (reemplazo → izq) ≠ NULL do                            ▷ Θ(log(n)) iteraciones
            reemplazo ← (reemplazo → izq)                            ▷ Θ(1)
        end while
        puntero(nodo) padreReemplazo ← (reemplazo → padre)           ▷ Θ(1)
        if padre ≠ NULL then                                         ▷ Θ(1)
            if padre → izq = lugar then                              ▷ Θ(1)
                (padre → izq) ← reemplazo                            ▷ Θ(1)
            else
                (padre → der) ← reemplazo                            ▷ Θ(1)
            end if
        else
            a.raiz ← reemplazo                                       ▷ Θ(1)
        end if
        (reemplazo → padre) ← padre                                  ▷ Θ(1)
        (reemplazo → izq) ← lugar → izq                              ▷ Θ(1)
        (lugar → izq → padre) ← reemplazo                            ▷ Θ(1)
        (padreReemplazo → izq) ← (reemplazo → der)                   ▷ Θ(1)
        if (reemplazo → der) ≠ NULL then                             ▷ Θ(1)
            (reemplazo → der → padre) ← padreReemplazo               ▷ Θ(1)
        end if
        (reemplazo → der) ← (lugar → der)                            ▷ Θ(1)
        (lugar → der → padre) ← reemplazo                            ▷ Θ(1)
        RebalancearArbol(reemplazo)                                  ▷ Θ(log(n))
    end if
end if
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 $O(\log(n))$.

iBuscar(**in** a : `estrAVL`, **in** k : κ, **out** $padre$: `estrAVL`) $\rightarrow$ res : `puntero(estrAVL)`

$padre \leftarrow$ Nil() $\triangleright\ \Theta(1)$
$actual \leftarrow a$ $\triangleright\ \Theta(1)$
while $\neg$ EsNil($actual$) $\wedge_{\text{L}}$ (Raiz($actual$).$clave \neq k$) **do** $\triangleright\ \Theta(\log(n))$ iteraciones
  $padre \leftarrow actual$ $\triangleright\ \Theta(1)$
  if $k \leq$ Raiz($actual$).$clave$ **then** $\triangleright\ \Theta(order(k))$
    $actual \leftarrow$ Izq($actual$) $\triangleright\ \Theta(1)$
  else
    $actual \leftarrow$ Der($actual$) $\triangleright\ \Theta(1)$
  end if
end while
$res \leftarrow actual$ $\triangleright\ \Theta(1)$

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 $\Theta(order(k))$.

iRecalcularAltura(**in** n : `puntero(nodo)`)

if $n \rightarrow izq \neq$ NULL $\wedge$ $n \rightarrow der \neq$ NULL **then** $\triangleright\ \Theta(1)$
  $(n \rightarrow altSubarbol) \leftarrow 1 + \max(n \rightarrow izq \rightarrow altSubarbol, n \rightarrow der \rightarrow altSubarbol)$ $\triangleright\ \Theta(1)$
else if $n \rightarrow izq \neq$ NULL **then** $\triangleright\ \Theta(1)$
  $(n \rightarrow altSubarbol) \leftarrow 1 + (n \rightarrow izq \rightarrow altSubarbol)$ $\triangleright\ \Theta(1)$
else if $n \rightarrow der \neq$ NULL **then** $\triangleright\ \Theta(1)$
  $(n \rightarrow altSubarbol) \leftarrow 1 + (n \rightarrow der \rightarrow altSubarbol)$ $\triangleright\ \Theta(1)$
else
  $(n \rightarrow altSubarbol) \leftarrow 1$ $\triangleright\ \Theta(1)$
end if

Descripción: Esta operación privada recibe un puntero a un nodo del árbol 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 $\Theta(1)$.

iFactorDeBalanceo(**in** n : `puntero(nodo)`) $\rightarrow$ res : `int`

`int` $altIzq \leftarrow n \rightarrow izq =$ NULL ? 0 : $n \rightarrow izq \rightarrow altSubarbol$ $\triangleright\ \Theta(1)$
`int` $altDer \leftarrow n \rightarrow der =$ NULL ? 0 : $n \rightarrow izq \rightarrow altSubarbol$ $\triangleright\ \Theta(1)$
$res \leftarrow altDer - altIzq$ $\triangleright\ \Theta(1)$

Descripción: Esta operación privada recibe un puntero a un nodo del árbol y calcula su factor de balanceo.

Complejidad: $\Theta(1)$

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

iRotarAIzquierda(**in** n : puntero(nodo))

if $n.padre \neq$ NULL **then** $\triangleright\ \Theta(1)$
  if $n.padre \rightarrow izq = n$ **then** $\triangleright\ \Theta(1)$
    $(n \rightarrow padre \rightarrow izq) \leftarrow n \rightarrow der$ $\triangleright\ \Theta(1)$
  else
    $(n \rightarrow padre \rightarrow der) \leftarrow n \rightarrow der$ $\triangleright\ \Theta(1)$
  end if
end if
$(n \rightarrow der \rightarrow padre) \leftarrow n \rightarrow padre$ $\triangleright\ \Theta(1)$
$n \rightarrow padre \leftarrow n \rightarrow der$ $\triangleright\ \Theta(1)$
$n \rightarrow der \leftarrow (n \rightarrow der \rightarrow izq)$ $\triangleright\ \Theta(1)$
if $n \rightarrow der \neq$ NULL **then** $\triangleright\ \Theta(1)$
  $(n \rightarrow der \rightarrow padre) \leftarrow n$ $\triangleright\ \Theta(1)$
end if
$(n \rightarrow padre \rightarrow izq) \leftarrow n$ $\triangleright\ \Theta(1)$
RecalcularAltura(n) $\triangleright\ \Theta(1)$
RecalcularAltura($n \rightarrow padre$) $\triangleright\ \Theta(1)$

Descripción: Esta operación privada recibe un puntero a un nodo del árbol y realiza una rotación a izquierda de dicho nodo. ¡Ojo, rompe el invariante de representación! (Los campos *altSubarbol* de los nodos superiores quedan inconsistentes).

Complejidad: $\Theta(1)$

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

iRotarADerecha(**in** n : puntero(nodo))

if $n \rightarrow padre \neq$ NULL **then** $\triangleright\ \Theta(1)$
  if $n \rightarrow padre \rightarrow izq = n$ **then** $\triangleright\ \Theta(1)$
    $(n \rightarrow padre \rightarrow izq) \leftarrow n \rightarrow izq$ $\triangleright\ \Theta(1)$
  else
    $(n \rightarrow padre \rightarrow der) \leftarrow n \rightarrow izq$ $\triangleright\ \Theta(1)$
  end if
end if
$(n \rightarrow izq \rightarrow padre) \leftarrow n \rightarrow padre$ $\triangleright\ \Theta(1)$
$n \rightarrow padre \leftarrow n \rightarrow izq$ $\triangleright\ \Theta(1)$
$n \rightarrow izq \leftarrow (n \rightarrow izq \rightarrow der)$ $\triangleright\ \Theta(1)$
if $n \rightarrow izq \neq$ NULL **then** $\triangleright\ \Theta(1)$
  $(n \rightarrow izq \rightarrow padre) \leftarrow n$ $\triangleright\ \Theta(1)$
end if
$(n \rightarrow padre \rightarrow der) \leftarrow n$ $\triangleright\ \Theta(1)$
RecalcularAltura(n) $\triangleright\ \Theta(1)$
RecalcularAltura($n \rightarrow padre$) $\triangleright\ \Theta(1)$

Descripción: Esta operación privada recibe un puntero a un nodo del árbol y realiza una rotación a derecha de dicho nodo. ¡Ojo, rompe el invariante de representación! (Los campos *altSubarbol* de los nodos superiores quedan inconsistentes).

Complejidad: $\Theta(1)$

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

iRebalancearArbol(**in** n : `puntero(nodo)`)

```
puntero(nodo) p ← n                                   ▷ Θ(1)
while p ≠ NULL do                                      ▷ Θ(log(n)) iteraciones
    RecalcularAltura(p)                                ▷ Θ(1)
    int fdb1 ← FactorDeBalanceo(p)                     ▷ Θ(1)
    if fdb1 = 2 then                                   ▷ Θ(1)
        puntero(nodo) q ← (p → der)                    ▷ Θ(1)
        int fdb2 ← FactorDeBalanceo(q)                 ▷ Θ(1)
        if fdb2 = 1 ∨ fdb2 = 0 then                    ▷ Θ(1)
            RotarAIzquierda(p)                         ▷ Θ(1)
            p ← q                                      ▷ Θ(1)
        else if fdb2 = −1 then                         ▷ Θ(1)
            RotarADerecha(q)                           ▷ Θ(1)
            RotarAIzquierda(p)                         ▷ Θ(1)
            p ← (q → padre)                            ▷ Θ(1)
        end if
    else if fdb1 = −2 then
        puntero(nodo) q ← (p → izq)                    ▷ Θ(1)
        int fdb2 ← FactorDeBalanceo(q)                 ▷ Θ(1)
        if fdb2 = −1 ∨ fdb2 = 0 then                   ▷ Θ(1)
            RotarADerecha(p)                           ▷ Θ(1)
            p ← q                                      ▷ Θ(1)
        else if fdb2 = 1 then                          ▷ Θ(1)
            RotarAIzquierda(q)                         ▷ Θ(1)
            RotarADerecha(p)                           ▷ Θ(1)
            p ← (q → padre)                            ▷ Θ(1)
        end if
    end if
    p ← (p → padre)                                    ▷ Θ(1)
end while
```

Descripción: Esta operación privada recibe un puntero a un nodo del árbol y restaura el invariante de representación en la rama ascendente a partir de dicho nodo, 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 $\Theta(1)$.