

# 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

<b>1. Módulo Red</b>	<b>5</b>
<b>2. Módulo Árbol binario</b>	<b>14</b>
<b>3. Módulo Diccionario Logarítmico</b>	<b>18</b>



# 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**  $\equiv \{res =_{obs} iniciarRed()\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Genera una nueva red sin ninguna computadora.

AGREGARCOMPU(**in/out**  $r : Red$ , **in**  $c : compu$ )

**Pre**  $\equiv \{r =_{obs} r_0 \wedge (\forall c' : compu)(c' \in computadoras(r) \rightarrow ip(c) \neq ip(c'))\}$

**Post**  $\equiv \{r =_{obs} agregarCompu(r_0, c)\}$

**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**  $\equiv \{r =_{obs} r_0 \wedge c_1 \in computadoras(r) \wedge c_2 \in computadoras(r) \wedge ip(c_0) \neq ip(c_1) \wedge \neg conectadas?(r, c_0, c_1) \wedge \neg usaInterfaz?(r, c_0, i_0) \wedge \neg usaInterfaz?(r, c_1, i_1)\}$

**Post**  $\equiv \{r =_{obs} conectar(r_0, c_0, i_0, c_1, i_1)\}$

**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**  $\equiv \{res =_{obs} conectadas?(r, c_0, c_1)\}$

**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**  $\equiv \{c_0 \in computadoras(r) \wedge c_1 \in computadoras(r) \wedge_L conectadas?(r, c_0, c_1)\}$

**Post**  $\equiv \{res =_{obs} interfazUsada(r, c_0, c_1)\}$

**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  $r : \text{Red}$ , in  $c : \text{compu}$ , in  $i : \text{interfaz}$ )  $\rightarrow res : \text{bool}$

**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  $r : \text{Red}$ , in  $c_0 : \text{compu}$ , in  $c_1 : \text{compu}$ )  $\rightarrow res : \text{conj}(\text{secu}(\text{compu}))$

**Pre**  $\equiv \{c_0 \in \text{computadoras}(r) \wedge c_1 \in \text{computadoras}(r)\}$

**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 mínimos posibles entre  $c_0$  y  $c_1$ . De no haber ninguno, devuelve  $\emptyset$ .

**Aliasing:** Devuelve el conjunto por copia.

**HAYCAMINO?**(in  $r : \text{Red}$ , in  $c_0 : \text{compu}$ , in  $c_1 : \text{compu}$ )  $\rightarrow res : \text{bool}$

**Pre**  $\equiv \{c_0 \in \text{computadoras}(r) \wedge c_1 \in \text{computadoras}(r)\}$

**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  $r : \text{Red}$ )  $\rightarrow res : \text{nat}$

**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  $r : \text{Red}$ )  $\rightarrow res : \text{Red}$

**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  $\rightarrow$  bool`

$\text{Rep}(e) \equiv$

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

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

$\text{Abs}(e) \equiv (\text{r: Red} \mid \text{computadoras}(\text{r}) = e.\text{compus} \wedge$   
 $(\forall c_1: \text{compu}) ((\forall c_2: \text{compu}) \text{conectados?}(\text{r}, c_1, c_2) = \text{esta?}(c_2, \text{JuntarSignificados}(\text{Obtener}(c_1,$   
 $e.\text{conexiones})))) \wedge$   
 $(\forall c_1: \text{compu}) ((\forall c_2: \text{compu}) \text{InterfazUsada}(\text{r}, c_1, c_2) = \text{DameClave}(\text{Obtener}(c_1, e.\text{conexiones}), c_2)))$

### Funciones Auxiliares:

$\text{sinRepetidos} : \text{secu}(\text{nat}) \longrightarrow \text{bool}$

$\text{sinRepetidos}(s) \equiv \#(\text{pasarSecuAConj}(s)) = \text{long}(s)$

$\text{pasarSecuAConj} : \text{secu}(\text{nat}) \longrightarrow \text{conj}(\text{nat})$

$\text{pasarSecuAConj}(s) \equiv \text{if } \text{vacía?}(s) \text{ then } \emptyset \text{ else } \text{Ag}(\text{prim}(s), \text{pasarSecuAConj}(\text{fin}(s))) \text{ fi}$

$\text{JuntarSignificados} : \text{dicc}(\text{Interfaz}, \text{IP}) \longrightarrow \text{secu}(\text{IP})$

$\text{JuntarSignificados}(d) \equiv \text{JuntarSignificados}_{\text{aux}}(d, \text{claves}(d))$

$\text{JuntarSignificados}_{\text{aux}} : \text{dicc}(\text{Interfaz}, \text{IP}) \times \text{conj}(\text{Interfaz}) \longrightarrow \text{secu}(\text{IP})$

$\text{JuntarSignificados}_{\text{aux}}(d, Is) \equiv \text{if } \emptyset?(Is) \text{ then}$   
 $\quad \langle \rangle$   
 $\quad \text{else}$   
 $\quad \quad \text{obtener}(\text{dameUno}(Is, d)) \bullet \text{JuntarSignificados}_{\text{aux}}(d, \text{sinUno}(Is))$   
 $\quad \text{fi}$

$\text{DameClave} : \text{dicc}(\text{Interfaz}, \text{IP}) d \times \text{IP } c \longrightarrow \text{Interfaz} \quad \{\text{esta?}(c, \text{JuntarSignificados}(d))\}$

$\text{DameClave}(d, c) \equiv \text{DameClave}_{\text{aux}}(d, \text{claves}(d), c)$

$\text{DameClave}_{\text{aux}} : \text{dicc}(\text{Interfaz}, \text{IP}) d \times \text{conj}(\text{Interfaz}) Is \times \text{IP } c \longrightarrow \text{Interfaz} \quad \{\text{esta?}(c, \text{JuntarSignificados}(d))\}$

$\text{DameClave}_{\text{aux}}(d, Is, c) \equiv \text{if } (\text{obtener}(\text{dameUno}(Is)) = c) \text{ then}$   
 $\quad \text{dameUno}(Is)$   
 $\quad \text{else}$   
 $\quad \quad \text{DameClave}_{\text{aux}}(d, \text{sinUno}(Is), c)$   
 $\quad \text{fi}$

## Algoritmos



### Algoritmo modificado

---

INICIARRED() $\rightarrow$ <i>res</i> : <b>estrRed</b>	
1 <i>res</i> $\leftarrow$ $\langle$ Vacio(), Vacio() $\rangle$	$\triangleright \Theta(1)$

---

**Complejidad:**  $\Theta(1)$



### Algoritmo modificado

---

IAgregarCompu( <b>in/out</b> <i>r</i> : <b>estrRed</b> , <b>in</b> <i>c</i> : <b>compu</b> )	
1 AgregarRapido( <i>r.compus</i> , <i>c</i> )	$\triangleright \Theta(1)$
2 DefinirRapido( <i>r.conexiones</i> , <i>c.IP</i> , Vacio())	$\triangleright \Theta(1)$

---

**Complejidad:**  $\Theta(1)$



### Algoritmo modificado

---

ICONECTAR( <b>in/out</b> <i>r</i> : <b>estrRed</b> ), <b>in</b> <i>c</i> <sub>1</sub> : <b>compu</b> , <b>in</b> <i>i</i> <sub>1</sub> : <b>interfaz</b> , <b>in</b> <i>c</i> <sub>2</sub> : <b>compu</b> , <b>in</b> <i>i</i> <sub>2</sub> : <b>interfaz</b> )	
1 itDicc(IP, diccConexiones) <i>it</i> <sub>1</sub> $\leftarrow$ CrearIt( <i>r.conexiones</i> )	$\triangleright \Theta(1)$
2 itDicc(IP, diccConexiones) <i>it</i> <sub>2</sub> $\leftarrow$ CrearIt( <i>r.conexiones</i> )	$\triangleright \Theta(1)$
3 <b>while</b> SiguieteClave( <i>it</i> <sub>1</sub> ) $\neq$ <i>c</i> <sub>1</sub> .IP <b>do</b>	$\triangleright \Theta(n)$ iteraciones
4     Avanzar( <i>it</i> <sub>1</sub> )	$\triangleright \Theta(1)$
5 <b>end while</b>	
6 <b>while</b> SiguieteClave( <i>it</i> <sub>2</sub> ) $\neq$ <i>c</i> <sub>1</sub> .IP <b>do</b>	$\triangleright \Theta(n)$ iteraciones
7     Avanzar( <i>it</i> <sub>2</sub> )	$\triangleright \Theta(1)$
8 <b>end while</b>	
9 DefinirRapido(SiguieteSignificado( <i>it</i> <sub>1</sub> ), <i>i</i> <sub>1</sub> , Copiar( <i>it</i> <sub>2</sub> ))	$\triangleright \Theta(1)$
10 DefinirRapido(SiguieteSignificado( <i>it</i> <sub>2</sub> ), <i>i</i> <sub>2</sub> , Copiar( <i>it</i> <sub>1</sub> ))	$\triangleright \Theta(1)$

---

**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 modificado**


---

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


---

```

1 itDicc(IP, diccConexiones)  $it_1 \leftarrow \text{CrearIt}(r.conexiones)$   $\triangleright \Theta(1)$ 
2 while SiguienteClave( $it_1$ )  $\neq c_1.IP$  do  $\triangleright \Theta(n)$  iteraciones
3   | Avanzar( $it_1$ )  $\triangleright \Theta(1)$ 
4 end while
5 itDicc(interfaz, itDicc(IP, diccConexiones))  $it_2 \leftarrow \text{CrearIt}(\text{Significado}(\text{Siguiente}(it_1)))$   $\triangleright \Theta(1)$ 
6 while HaySiguiente?( $it_2$ )  $\wedge_L$  SiguienteClave(SiguienteSignificado( $it_2$ ))  $\neq c_2.IP$  do  $\triangleright \Theta(I)$  iteraciones
7   | Avanzar( $it_2$ )  $\triangleright \Theta(1)$ 
8 end while
9  $res \leftarrow \text{HaySiguiente?}(it_2) \wedge_L \text{SiguienteClave}(\text{SiguienteSignificado}(it_2)) = c_2.IP$   $\triangleright \Theta(1)$ 

```

---

**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 modificado**


---

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


---

```

1 itLista(estrCompu)  $it_1 \leftarrow \text{crearIt}(r.compues)$   $\triangleright \Theta(1)$ 
2 while siguiente( $it_1$ ).IP  $\neq c_1.IP$  do  $\triangleright \Theta(n)$  iteraciones
3   | avanzar( $it_1$ )  $\triangleright \Theta(1)$ 
4 end while
5 itLista(tupla(interfaz, itLista(estrCompu)))  $it_2 \leftarrow \text{crearIt}(\text{siguiente}(it_1).conexiones)$   $\triangleright \Theta(1)$ 
6 while (siguiente(siguiente( $it_2$ ).com)).IP  $\neq c_2.IP$  do  $\triangleright \Theta(I)$  iteraciones
7   | avanzar( $it_1$ )  $\triangleright \Theta(1)$ 
8 end while
9  $res \leftarrow \text{siguiente}(it_2).inter$   $\triangleright \Theta(1)$ 

```

---

**Complejidad:**  $\Theta(n + I)$ 


---

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

---

```

1  $res \leftarrow \text{vacio}()$   $\triangleright \Theta(1)$ 
2 itLista(estrComp)  $it_1 \leftarrow \text{crearIt}(r.compues)$   $\triangleright \Theta(1)$ 
3 while siguiente( $it_1$ ).IP  $\neq c.IP$  do  $\triangleright \Theta(n)$  iteraciones
4   | avanzar( $it_1$ )  $\triangleright \Theta(1)$ 
5 end while
6 itLista(tupla(interfaz, itLista(estrCompu)))  $it_2 \leftarrow \text{crearIt}(\text{siguiente}(it_1).conexiones)$   $\triangleright \Theta(1)$ 
7 while haySiguiente?( $it_2$ ) do  $\triangleright \Theta(n)$  iteraciones
8   if haySiguiente?(siguiente( $it_2$ ).com) then  $\triangleright \Theta(1)$ 
9     | agregar( $res$ , (siguiente(siguiente( $it_2$ ).com).IP,
10      crearConjunto(siguiente(siguiente( $it_2$ ).com).conexiones))))  $\triangleright \Theta(I^2)$ 
11   end if
12   avanzar( $it_2$ )  $\triangleright \Theta(1)$ 
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 c1: compu, in c2: 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))))
6   | ▷ Θ(n³ × 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)  $\rightarrow res$ : conj(lista(compu))

---

```

1 itConj(lista(compu)) it ← crearIt(c)                                ▷  $\Theta(1)$ 
2 res ← vacío()                                                         ▷  $\Theta(1)$ 
3 while haySiguiente(it) do                                           ▷  $\Theta(n!)$  iteraciones
4   if long(siguiente(it)) =  $n$  then                                   ▷  $\Theta(1)$ 
5     | agregar(res, siguiente(it))                                   ▷  $\Theta(n)$ 
6     | avanzar(it)                                                  ▷  $\Theta(1)$ 
7   else
8     | avanzar(it)                                                  ▷  $\Theta(1)$ 
9   end if
10 end while

```

---

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

---

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

---

```

1 nat  $i$  ←  $n$                                                          ▷  $\Theta(1)$ 
2 itConj(lista(compu)) it ← crearIt(c)                                ▷  $\Theta(1)$ 
3 while haySiguiente(it) do
4   if long(siguiente(it)) then
5     |  $i$  ← longitud(siguiente(it))                                   ▷  $\Theta(1)$ 
6     | avanzar(it)                                                  ▷  $\Theta(1)$ 
7   else
8     | avanzar(it)                                                  ▷  $\Theta(1)$ 
9   end if
10 end while
11 res ←  $i$                                                            ▷  $\Theta(1)$ 

```

---

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

**Justificación:** Devuelve la longitud de la secuencia más chica

---

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

---

```

1 res ← vacío()                                                         ▷  $\Theta(1)$ 
2 ItConj it ← crearIt(c)                                              ▷  $\Theta(1)$ 
3 while haySiguiente(it) do                                           ▷  $\Theta(n)$  iteraciones
4   | agregarAtras(res, siguiente(it))                               ▷  $\Theta(I)$ 
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  $c_1$ : compu, in  $c_2$ : compu)  $\rightarrow res$ : bool

---

```

1 res ← (¬esVacio?(iCaminosMinimos( $r$ ,  $c_1$ ,  $c_2$ )))                    ▷  $\Theta(n^2 \times n!)$ 

```

---

**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))),
9         | caminos(r, c0, c1, l, fin(vec)))
10    | else
11      | res ← caminos(r, c0, c1, l, fin(vec))
12    | end if
13  end if
14 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()                                ▷  $\Theta(1)$ 
2 if vacio?(c1) then                            ▷  $\Theta(1)$ 
3   | res ← c2                                ▷  $\Theta(I \times n \times n!)$ 
4 else
5   | itConj(lista(compu)) it ← crearIt(c1)    ▷  $\Theta(1)$ 
6   | while haySiguiente(it) do                ▷  $\Theta(n)$ 
7     | Ag(siguiente(it), c2)                 ▷  $\Theta(n)$ 
8     | avanzar(it)                           ▷  $\Theta(1)$ 
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                            ▷  $\Theta(1)$ 
2   | res ← false                             ▷  $\Theta(1)$ 
3 else
4   | ItLista(compu) it ← crearIt(l)           ▷  $\Theta(1)$ 
5   | while haySiguiente(it) ∧ siguiente(it) ≠ c do ▷  $\Theta(n)$ 
6     |
7     | | avanzar(it)                         ▷  $\Theta(1)$ 
8     | | end while
9   | end if
10 res ← (haySiguiente(it))                    ▷  $\Theta(1)$ 
```

---

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

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

**Justificación:** .

---

ICOMPUTADORAS(**in**  $r : \text{estrRed}$ )  $\rightarrow res : \text{conj}(\text{compu})$ 


---

```

1  $res \leftarrow \text{vacío}()$   $\triangleright \Theta(1)$ 
2 itRed  $it \leftarrow \text{crearItRed}()$   $\triangleright \Theta(1)$ 
3 while haySiguiente?( $it$ ) do  $\triangleright \Theta(n)$  iteraciones
4   | agregar( $res$ , siguiente( $it$ ))  $\triangleright \Theta(n + I^2)$ 
5   | avanzar( $it$ )  $\triangleright \Theta(1)$ 
6 end while
```

---

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

---

ICOPIAR(**in**  $r : : \text{estrRed}$ )  $\rightarrow res : \text{Red}$ 


---

```

1  $res \leftarrow \langle \text{copiar}(r.\text{compus}, r.\text{cantidadCompus}) \rangle$   $\triangleright \Theta(n \times I)$ 
```

---

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

**Justificación:** Devuelve una copia de la Red

---

ICANTCOMPUS(**in**  $r : \text{Red}$ )  $\rightarrow res : \text{nat}$ 


---

```

1  $res \leftarrow r.\text{cantCompus}$   $\triangleright \Theta(1)$ 
```

---

**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**       $\alpha$

**se explica con:**     $\text{ARBOL\_BINARIO}(\alpha)$

**géneros:**             $\text{ab}(\alpha)$

### Operaciones del TAD Árbol binario

$\text{NIL}() \rightarrow res : \text{ab}(\alpha)$

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

**Post**  $\equiv \{res =_{\text{obs}} \text{nil}\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Crea y devuelve un árbol binario vacío.

$\text{BIN}(\text{in } i : \text{ab}(\alpha), \text{in } r : \alpha, \text{in } d : \text{ab}(\alpha)) \rightarrow res : \text{ab}(\alpha)$

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

**Post**  $\equiv \{res =_{\text{obs}} \text{bin}(i, r, d) \wedge \text{esAlias}(\text{izq}(res), i) \wedge \text{esAlias}(\text{raiz}(res), r) \wedge \text{esAlias}(\text{der}(res), d)\}$

**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.

$\text{ESNIL}(\text{in } a : \text{ab}(\alpha)) \rightarrow res : \text{bool}$

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

**Post**  $\equiv \{res =_{\text{obs}} \text{nil?}(a)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve *true* si y solo si el árbol binario está vacío.

$\text{RAIZ}(\text{in } a : \text{ab}(\alpha)) \rightarrow res : \alpha$

**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.

$\text{IZQ}(\text{in } a : \text{ab}(\alpha)) \rightarrow res : \alpha$

**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.

$\text{DER}(\text{in } a : \text{ab}(\alpha)) \rightarrow res : \alpha$

**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**  $a : \text{ab}(\alpha)$ )  $\rightarrow res : \text{nat}$

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

**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**  $a : \text{ab}(\alpha)$ )  $\rightarrow res : \text{nat}$

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

**Post**  $\equiv \{res =_{\text{obs}} \text{tamaño}(a)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve la cantidad de nodos del árbol binario.

**INORDER**(**in**  $a : \text{ab}(\alpha)$ )  $\rightarrow res : \text{lista}(\alpha)$

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

**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**  $a : \text{ab}(\alpha)$ )  $\rightarrow res : \text{lista}(\alpha)$

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

**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**  $a : \text{ab}(\alpha)$ )  $\rightarrow res : \text{lista}(\alpha)$

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

**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

$\text{ab}(\alpha)$  se representa con puntero(nodo)

donde nodo es  $\text{tupla}(\text{valor} : \alpha,$   
 $\text{izq} : \text{puntero}(\text{nodo}),$   
 $\text{der} : \text{puntero}(\text{nodo}))$

$\text{Rep} : \text{puntero}(\text{nodo}) \rightarrow \text{bool}$

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

$\text{Abs} : \text{puntero}(\text{nodo}) \rightarrow \text{ab}(\alpha)$

$\{\text{Rep}(a)\}$

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

$\text{hijos} : \text{nodo} \rightarrow \text{conj}(\text{nodo})$

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

$\text{nodos} : \text{puntero}(\text{nodo}) \rightarrow \text{conj}(\text{nodo})$

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

altura : puntero(nodo)  $\rightarrow$  nat

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

padres : puntero(nodo)  $\times$  conj(nodo)  $\rightarrow$  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	$res \leftarrow \text{NULL}$	$\triangleright \Theta(1)$
<b>Complejidad:</b> $\Theta(1)$		
iBIN(in $i$ : puntero(nodo), in $r$ : $\alpha$ , in $d$ : puntero(nodo)) $\rightarrow res$ : puntero(nodo)		
1	$res \leftarrow \&\langle r, i, d, 1 + \max(\text{Altura}(i), \text{Altura}(d)), 1 + \text{CantNodos}(i) + \text{CantNodos}(d) \rangle$ // Reservamos memoria para el nuevo nodo	$\triangleright \Theta(1)$
<b>Complejidad:</b> $\Theta(1)$		
iEsNIL(in $a$ : $\text{ab}(\alpha)$ ) $\rightarrow res$ : bool		
1	$res \leftarrow a = \text{NULL}$	$\triangleright \Theta(1)$
<b>Complejidad:</b> $\Theta(1)$		
iRAIZ(in $a$ : $\text{ab}(\alpha)$ ) $\rightarrow res$ : $\alpha$		
1	$res \leftarrow a \rightarrow \text{valor}$	$\triangleright \Theta(1)$
<b>Complejidad:</b> $\Theta(1)$		
iIzQ(in $a$ : $\text{ab}(\alpha)$ ) $\rightarrow res$ : $\text{ab}(\alpha)$		
1	$res \leftarrow a \rightarrow \text{izq}$	$\triangleright \Theta(1)$
<b>Complejidad:</b> $\Theta(1)$		
iDER(in $a$ : $\text{ab}(\alpha)$ ) $\rightarrow res$ : $\text{ab}(\alpha)$		
1	$res \leftarrow a \rightarrow \text{der}$	$\triangleright \Theta(1)$
<b>Complejidad:</b> $\Theta(1)$		
iALTURA(in $a$ : $\text{ab}(\alpha)$ ) $\rightarrow res$ : nat		
1	<b>if</b> EsNil( $a$ ) <b>then</b>	$\triangleright \Theta(1)$
2	$res \leftarrow 0$	$\triangleright \Theta(1)$
3	<b>else</b>	
4	$res \leftarrow 1 + \max(\text{Altura}(\text{Izq}(a)), \text{Altura}(\text{Der}(a)))$	$\triangleright \Theta(n)$ (ver justificación)
5	<b>end if</b>	
<b>Complejidad:</b> $\Theta(n)$		



**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  $\Theta(n)$ .

---

$\text{ICANTNODOS}(\text{in } a : \text{ab}(\alpha)) \rightarrow res : \text{nat}$	
1 <b>if</b> EsNil( $a$ ) <b>then</b>	$\triangleright \Theta(1)$
2     $res \leftarrow 0$	$\triangleright \Theta(1)$
3 <b>else</b>	
4     $res \leftarrow 1 + \text{CantNodos}(\text{lzq}(a)) + \text{CantNodos}(\text{Der}(a))$	$\triangleright \Theta(n)$ (ver justificación)
5 <b>end if</b>	

---

**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

<b>géneros</b>	$\kappa, \sigma$
<b>función</b>	$\bullet = \bullet(\text{in } k_1 : \kappa, \text{in } k_2 : \kappa) \rightarrow res : \text{bool}$ <b>Pre</b> $\equiv \{\text{true}\}$ <b>Post</b> $\equiv \{res =_{\text{obs}} (k_1 = k_2)\}$ <b>Complejidad:</b> $\Theta(\text{equal}(k_1, k_2))$ <b>Descripción:</b> función de igualdad de $\kappa$
<b>función</b>	$\bullet \leq \bullet(\text{in } k_1 : \kappa, \text{in } k_2 : \kappa) \rightarrow res : \text{bool}$ <b>Pre</b> $\equiv \{\text{true}\}$ <b>Post</b> $\equiv \{res =_{\text{obs}} (k_1 \leq k_2)\}$ <b>Complejidad:</b> $\Theta(\text{order}(k_1, k_2))$ <b>Descripción:</b> función de comparación por orden total estricto de $\kappa$
<b>función</b>	<b>COPIAR</b> ( <b>in</b> $k : \kappa$ ) $\rightarrow res : \kappa$ <b>Pre</b> $\equiv \{\text{true}\}$ <b>Post</b> $\equiv \{res =_{\text{obs}} k\}$ <b>Complejidad:</b> $\Theta(\text{copy}(k))$ <b>Descripción:</b> función de copia de $\kappa$
<b>función</b>	<b>COPIAR</b> ( <b>in</b> $s : \sigma$ ) $\rightarrow res : \sigma$ <b>Pre</b> $\equiv \{\text{true}\}$ <b>Post</b> $\equiv \{res =_{\text{obs}} s\}$ <b>Complejidad:</b> $\Theta(\text{copy}(s))$ <b>Descripción:</b> función de copia de $\sigma$
<b>se explica con:</b>	<b>DICCIONARIO</b> ( $\kappa, \sigma$ )
<b>géneros:</b>	<b>diccLog</b> ( $\kappa, \sigma$ )

#### Operaciones de diccionario

**VACIO**()  $\rightarrow res : \text{diccLog}(\kappa, \sigma)$   
**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**  $d : \text{diccLog}(\kappa, \sigma)$ , **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:**  $\Theta(\log(n) \times \text{order}(k) + \text{copy}(k) + \text{copy}(s))$   
**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, sobrescribe su significado con el nuevo.  
**Aliasing:** Las claves y significados se almacenan por copia.

**BORRAR**(**in/out**  $d : \text{diccLog}(\kappa, \sigma)$ , **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**  $d: \text{diccLog}(\kappa, \sigma) \rightarrow res : \text{nat}$

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

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

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

**Descripción:** Devuelve la cantidad de claves del diccionario.

DEFINIDO?(**in**  $d: \text{diccLog}(\kappa, \sigma)$ , **in**  $k: \kappa \rightarrow res : \text{bool}$

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

**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**  $d: \text{diccLog}(\kappa, \sigma)$ , **in**  $k: \kappa \rightarrow res : \sigma$

**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

$\text{diccLog}(\kappa, \sigma)$  se representa con **estrAVL**

donde **estrAVL** es  $\text{ab}(\text{tupla}(\text{clave}: \kappa, \text{significado}: \sigma, \text{altSubarbol}: \text{nat}, \text{padre}: \text{estrAVL}))$

$\text{Rep} : \text{estrAVL} \rightarrow \text{bool}$

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

$\text{Abs} : \text{estrAVL } a \rightarrow \text{dicc}(\kappa, \sigma)$

$\{\text{Rep}(a)\}$

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

$\text{unir} : \text{dicc}(\kappa \times \sigma) \times \text{dicc}(\kappa \times \sigma) \rightarrow \text{dicc}(\kappa, \sigma)$

```

unir( $d_1, d_2$ )  $\equiv$  if vacío( $d_2$ ) then
     $d_1$ 
else
    definir(
        dameUno(claves( $d_2$ )),
        obtener(dameUno(claves( $d_2$ )),  $d_2$ ),
        unir( $d_1$ , borrar(dameUno(claves( $d_2$ )),  $d_2$ ))
    )
fi

```

## Algoritmos



### Algoritmo modificado

---

IVACIO()  $\rightarrow res : \text{estrAVL}$

---

1  $res \leftarrow \text{Nil}()$   $\triangleright \Theta(1)$

---

**Complejidad:**  $\Theta(1)$



### Algoritmo modificado

---

IDEFINIR(**in**  $k : \kappa$ , **in**  $s : \sigma$ , **in/out**  $a : \text{estrAVL}$ )

---

```

1 estrAVL padre
2 estrAVL lugar  $\leftarrow \text{Buscar}(a, k, \text{padre})$   $\triangleright \Theta(\log(n) \times \text{order}(k))$ 
3 if  $\neg \text{EsNil}(\text{lugar})$  then  $\triangleright \Theta(1)$ 
4   |  $\text{Raiz}(\text{lugar}).\text{significado} \leftarrow \text{Copiar}(s)$   $\triangleright \Theta(\text{copy}(s))$ 
5 else
6   | estrAVL nuevo  $\leftarrow \& \text{Bin}(\text{Nil}(), \langle \text{Copiar}(k), \text{Copiar}(s), 1, \text{padre} \rangle, \text{Nil}())$  // Reservamos memoria para el
   | nuevo nodo  $\triangleright \Theta(\text{copy}(k) + \text{copy}(s))$ 
7   | if  $k \leq \text{Raiz}(\text{padre}).\text{clave}$  then  $\triangleright \Theta(\text{order}(k))$ 
8     |  $\text{padre} \leftarrow \text{Bin}(\text{nuevo}, \text{Raiz}(\text{padre}), \text{Der}(\text{padre}))$   $\triangleright \Theta(1)$ 
9   | else
10    |  $\text{padre} \leftarrow \text{Bin}(\text{Izq}(\text{padre}), \text{Raiz}(\text{padre}), \text{nuevo})$   $\triangleright \Theta(1)$ 
11    end if
12     $\text{RebalancearArbol}(\text{padre})$   $\triangleright \Theta(\log(n))$ 
13 end if

```

---

**Complejidad:**  $\Theta(\log(n) \times \text{order}(k) + \text{copy}(k) + \text{copy}(s))$

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



### Algoritmo modificado

---

IOTENER(**in**  $a : \text{estrAVL}$ , **in**  $k : \kappa$ )  $\rightarrow res : \sigma$

---

```

1 estrAVL padre
2 estrAVL lugar  $\leftarrow \text{Buscar}(a, k, \text{padre})$   $\triangleright \Theta(\log(n) \times \text{order}(k))$ 
3  $res \leftarrow \text{Raiz}(\text{lugar}).\text{significado}$   $\triangleright \Theta(1)$ 

```

---

**Complejidad:**  $\Theta(\log(n) \times \text{order}(k))$

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

**Algoritmo modificado**


---

 $\text{ICANTCLAVES}(\text{in } a : \text{estrAVL}) \rightarrow res : \text{nat}$ 


---

 $1 \text{ } res \leftarrow \text{CantNodos}(a)$ 
 $\triangleright \Theta(n)$ 


---

**Complejidad:**  $\Theta(n)$ **Algoritmo modificado**


---

 $\text{IDEFINIDO?}(\text{in } a : \text{estrAVL}, \text{in } k : \kappa) \rightarrow res : \text{bool}$ 


---

 $1 \text{ } \text{estrAVL } padre$ 
 $2 \text{ } \text{estrAVL } lugar \leftarrow \text{Buscar}(a, k, padre)$ 
 $\triangleright \Theta(\log(n) \times order(k))$ 
 $3 \text{ } res \leftarrow \neg \text{EsNil}(lugar)$ 
 $\triangleright \Theta(1)$ 


---

**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  $k$ :  $\kappa$ )**


---

```

1  estrAVL padre
2  estrAVL lugar  $\leftarrow$  Buscar( $a$ ,  $k$ , padre)  $\triangleright \Theta(\log(n) \times \text{order}(k))$ 
3  if EsNil(lzq(lugar))  $\wedge$  EsNil(Der(lugar)) then  $\triangleright \Theta(1)$ 
4      if  $\neg$  EsNil(padre) then  $\triangleright \Theta(1)$ 
5          if lzq(padre) = lugar then  $\triangleright \Theta(1)$ 
6              | padre  $\leftarrow$  Bin(Nil(), Raiz(padre), Der(padre))  $\triangleright \Theta(1)$ 
7          else
8              | padre  $\leftarrow$  Bin(lzq(padre), Raiz(padre), Nil())  $\triangleright \Theta(1)$ 
9          end if
10         RebalancearArbol(padre)  $\triangleright \Theta(\log(n))$ 
11     else
12         |  $a \leftarrow$  Nil()  $\triangleright \Theta(1)$ 
13     end if
14 else if EsNil(Der(lugar)) then  $\triangleright \Theta(1)$ 
15     Raiz(lzq(lugar)).padre  $\leftarrow$  padre  $\triangleright \Theta(1)$ 
16     if  $\neg$  EsNil(padre) then  $\triangleright \Theta(1)$ 
17         if lzq(padre) = lugar then  $\triangleright \Theta(1)$ 
18             | padre  $\leftarrow$  Bin(lzq(lugar), Raiz(padre), Der(padre))  $\triangleright \Theta(1)$ 
19         else
20             | padre  $\leftarrow$  Bin(lzq(padre), Raiz(padre), lzq(lugar))  $\triangleright \Theta(1)$ 
21         end if
22         RebalancearArbol(padre)  $\triangleright \Theta(\log(n))$ 
23     else
24         |  $a \leftarrow$  lzq(lugar)  $\triangleright \Theta(1)$ 
25     end if
26 else if EsNil(lzq(lugar)) then  $\triangleright \Theta(1)$ 
27     Raiz(lzq(lugar)).padre  $\leftarrow$  padre  $\triangleright \Theta(1)$ 
28     if  $\neg$  EsNil(padre) then  $\triangleright \Theta(1)$ 
29         if lzq(padre) = lugar then  $\triangleright \Theta(1)$ 
30             | padre  $\leftarrow$  Bin(Der(lugar), Raiz(padre), Der(padre))  $\triangleright \Theta(1)$ 
31         else
32             | padre  $\leftarrow$  Bin(lzq(padre), Raiz(padre), Der(lugar))  $\triangleright \Theta(1)$ 
33         end if
34         RebalancearArbol(padre)  $\triangleright \Theta(\log(n))$ 
35     else
36         |  $a \leftarrow$  Der(lugar)  $\triangleright \Theta(1)$ 
37     end if

```

---

IBORRAR (cont.)

```

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

```

**Complejidad:**  $\Theta(\log(n) \times \text{order}(k))$ 

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

**Algoritmo modificado**


---

<b>IBUSCAR</b> (in $a$ : <b>estrAVL</b> , in $k$ : $\kappa$ , out $padre$ : <b>estrAVL</b> ) $\rightarrow$ $res$ : puntero( <b>estrAVL</b> )		
1	$padre \leftarrow Nil()$	$\triangleright \Theta(1)$
2	$actual \leftarrow a$	$\triangleright \Theta(1)$
3	<b>while</b> $\neg EsNil(actual) \wedge_L (Raiz(actual).clave \neq k)$ <b>do</b>	$\triangleright \Theta(\log(n))$ iteraciones
4	$padre \leftarrow actual$	$\triangleright \Theta(1)$
5	<b>if</b> $k \leq Raiz(actual).clave$ <b>then</b>	$\triangleright \Theta(order(k))$
6	$actual \leftarrow lzq(actual)$	$\triangleright \Theta(1)$
7	<b>else</b>	
8	$actual \leftarrow Der(actual)$	$\triangleright \Theta(1)$
9	<b>end if</b>	
10	<b>end while</b>	
11	$res \leftarrow actual$	$\triangleright \Theta(1)$

---

**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  $\Theta(order(k))$ .

**Algoritmo modificado**


---

<b>IRECALCULARALTURA</b> (in $a$ : <b>estrAVL</b> )		
1	<b>if</b> $\neg EsNil(lzq(a)) \wedge \neg EsNil(der(a))$ <b>then</b>	$\triangleright \Theta(1)$
2	$Raiz(a).altSubarbol \leftarrow 1 + \max(Raiz(lzq(a)).altSubarbol, Raiz(der(a)).altSubarbol)$	$\triangleright \Theta(1)$
3	<b>else if</b> $\neg EsNil(lzq(a))$ <b>then</b>	$\triangleright \Theta(1)$
4	$Raiz(a).altSubarbol \leftarrow 1 + Raiz(lzq(a)).altSubarbol$	$\triangleright \Theta(1)$
5	<b>else if</b> $\neg EsNil(der(a))$ <b>then</b>	$\triangleright \Theta(1)$
6	$Raiz(a).altSubarbol \leftarrow 1 + Raiz(der(a)).altSubarbol$	$\triangleright \Theta(1)$
7	<b>else</b>	
8	$Raiz(a).altSubarbol \leftarrow 1$	$\triangleright \Theta(1)$
9	<b>end if</b>	

---

**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 operaciones con complejidad  $\Theta(1)$ .

**Algoritmo modificado**


---

<b>IFACTORDEBALANCEO</b> (in $a$ : <b>estrAVL</b> ) $\rightarrow$ $res$ : int		
1	int $altIzq \leftarrow EsNil(lzq(a)) ? 0 : Raiz(lzq(a)).altSubarbol$	$\triangleright \Theta(1)$
2	int $altDer \leftarrow EsNil(der(a)) ? 0 : Raiz(der(a)).altSubarbol$	$\triangleright \Theta(1)$
3	$res \leftarrow altDer - altIzq$	$\triangleright \Theta(1)$

---



**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	<code>estrAVL nuevo<sub>1</sub> ← Bin(lzq(a), Raiz(a), lzq(Der(a)))</code>	▷ $\Theta(1)$
2	<code>estrAVL nuevo<sub>2</sub> ← Bin(nuevo<sub>1</sub>, Raiz(Der(a)), Der(Der(a)))</code>	▷ $\Theta(1)$
3	<code>delete(Der(a)) // Liberamos la memoria de los subárboles que ya no usaremos</code>	▷ $\Theta(1)$
4	<code>delete(a)</code>	▷ $\Theta(1)$
5	<code>a ← nuevo<sub>2</sub></code>	▷ $\Theta(1)$
6	<code>Raiz(a).padre ← Raiz(lzq(a)).padre</code>	▷ $\Theta(1)$
7	<code>Raiz(lzq(a)).padre ← a</code>	▷ $\Theta(1)$
8	<code>Raiz(Der(a)).padre ← a</code>	▷ $\Theta(1)$
9	<code>Raiz(lzq(lzq(a))).padre ← lzq(a)</code>	▷ $\Theta(1)$
10	<code>Raiz(Der(lzq(a))).padre ← lzq(a)</code>	▷ $\Theta(1)$
11	<code>RecalcularAltura(lzq(a))</code>	▷ $\Theta(1)$
12	<code>RecalcularAltura(a)</code>	▷ $\Theta(1)$

---

**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	<code>estrAVL nuevo<sub>1</sub> ← Bin(Der(lzq(a)), Raiz(a), Der(a))</code>	▷ $\Theta(1)$
2	<code>estrAVL nuevo<sub>2</sub> ← Bin(lzq(lzq(a)), Raiz(lzq(a)), nuevo<sub>1</sub>)</code>	▷ $\Theta(1)$
3	<code>delete(lzq(a)) // Liberamos la memoria de los subárboles que ya no usaremos</code>	▷ $\Theta(1)$
4	<code>delete(a)</code>	▷ $\Theta(1)$
5	<code>a ← nuevo<sub>2</sub></code>	▷ $\Theta(1)$
6	<code>Raiz(a).padre ← Raiz(Der(a)).padre</code>	▷ $\Theta(1)$
7	<code>Raiz(lzq(a)).padre ← a</code>	▷ $\Theta(1)$
8	<code>Raiz(Der(a)).padre ← a</code>	▷ $\Theta(1)$
9	<code>Raiz(lzq(Der(a))).padre ← Der(a)</code>	▷ $\Theta(1)$
10	<code>Raiz(Der(Der(a))).padre ← Der(a)</code>	▷ $\Theta(1)$
11	<code>RecalcularAltura(Der(a))</code>	▷ $\Theta(1)$
12	<code>RecalcularAltura(a)</code>	▷ $\Theta(1)$

---

**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 $p \leftarrow a$	$\triangleright \Theta(1)$
2	while $\neg \text{EsNil}(p)$ do	$\triangleright \Theta(\log(n))$ iteraciones
3	RecalcularAltura( $p$ )	$\triangleright \Theta(1)$
4	int $fdb1 \leftarrow \text{FactorDeBalanceo}(p)$	$\triangleright \Theta(1)$
5	if $fdb1 = 2$ then	$\triangleright \Theta(1)$
6	estrAVL $q \leftarrow \text{Der}(p)$	$\triangleright \Theta(1)$
7	int $fdb2 \leftarrow \text{FactorDeBalanceo}(q)$	$\triangleright \Theta(1)$
8	if $fdb2 = 1 \vee fdb2 = 0$ then	$\triangleright \Theta(1)$
9	RotarAlzquierda( $p$ )	$\triangleright \Theta(1)$
10	else if $fdb2 = -1$ then	$\triangleright \Theta(1)$
11	RotarADerecha( $q$ )	$\triangleright \Theta(1)$
12	RotarAlzquierda( $p$ )	$\triangleright \Theta(1)$
13	end if	
14	else if $fdb1 = -2$ then	$\triangleright \Theta(1)$
15	estrAVL $q \leftarrow \text{lzq}(p)$	$\triangleright \Theta(1)$
16	int $fdb2 \leftarrow \text{FactorDeBalanceo}(q)$	$\triangleright \Theta(1)$
17	if $fdb2 = -1 \vee fdb2 = 0$ then	$\triangleright \Theta(1)$
18	RotarADerecha( $p$ )	$\triangleright \Theta(1)$
19	else if $fdb2 = 1$ then	$\triangleright \Theta(1)$
20	RotarAlzquierda( $q$ )	$\triangleright \Theta(1)$
21	RotarADerecha( $p$ )	$\triangleright \Theta(1)$
22	end if	
23	end if	
24	$p \leftarrow \text{Raiz}(p).padre$	$\triangleright \Theta(1)$
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  $\Theta(1)$ .