Algoritmos y Estructuras de Datos II

Primer Cuatrimestre de 2016

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

Recuperatorio Trabajo Práctico 2

Diseño

Grupo 4

Integrante	LU	Correo electrónico
Borgna, Agustin	79/15	aborgna@dc.uba.ar
Salvador, Alejo	467/15	alelucmdp@hotmail.com
Tamborindeguy, Guido	584/13	guido@tamborindeguy.com.ar
Zdanovitch, Nikita	520/14	3hb.tch@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Renombres

Para mejorar la legibilidad del TP, usamos los siguientes renombres.

- *registro* es $diccLog(campo, dato)$
- *campo* es *string*

2. Dato

2.1. Interfaz

usa:

se explica con: Dato

géneros: dato

operaciones:

DATONAT(**in** n : *nat*) $\rightarrow$ *res*: **dato**
Pre $\equiv$ {`Nat?(n)`}
Post $\equiv$ {`res` $=_{\text{obs}}$ `datoNat(n)`}
Complejidad: $\mathcal{O}(1)$

DATOSTRING(**in** *str* : *string*) $\rightarrow$ *res*: **dato**
Pre $\equiv$ {`String?(n)`}
Post $\equiv$ {`res` $=_{\text{obs}}$ `datoString(str)`}
Complejidad: $\mathcal{O}(\text{copy}(string))$
Aliasing: true

ISNAT(**in** d : *dato*) $\rightarrow$ *res*: **bool**
Pre $\equiv$ {`true`}
Post $\equiv$ {`res` $\iff$ `Nat?(d)`}
Complejidad: $\mathcal{O}(1)$

ISSTRING(**in** d : *dato*) $\rightarrow$ *res*: **bool**
Pre $\equiv$ {`true`}
Post $\equiv$ {`res` $\iff$ `String?(d)`}
Complejidad: $\mathcal{O}(1)$

GETNAT(**in** d : *dato*) $\rightarrow$ *res*: **nat**
Pre $\equiv$ {`Nat?(d)`}
Post $\equiv$ {`res` $\iff$ `valorNat(d)`}
Complejidad: $\mathcal{O}(1)$

GETSTRING(**in** d : *dato*) $\rightarrow$ *res*: **string**
Pre $\equiv$ {`String?(d)`}
Post $\equiv$ {`res` $\iff$ `valorStr(d)`}
Complejidad: $\mathcal{O}(1)$

$\bullet == \bullet$(**in** d_1 : *dato*, **in** d_2 : *dato*) $\rightarrow$ *res*: **bool**
Pre $\equiv$ {`true`}
Post $\equiv$ {`res` $\iff$ **if** `Nat?(`d_1`)` **then** `Nat?(`d_2`)` $\wedge_L$ `valorNat(`d_1`)` $=_{\text{obs}}$ `valorNat(`d_2`)` **else** `String?(`d_2`)` $\wedge_L$ `valorString(`d_1`)` $=_{\text{obs}}$ `valorString(`d_2`)` **fi** }
Complejidad: $\mathcal{O}(\text{cmp}(string))$

- $>$ •(**in** d_1 : *dato*, **in** d_2 : *dato*) → *res*: **bool**
  Pre ≡ {`true`}
  Post ≡ {`res` ⟺ **if** `tipo?(`d_1`)` ≠ `tipo?(`d_2`)` **then**
  `tipo?(`d_2`)`
  else if `Nat?(`d_1`)` **then**
  `valorNat(`d_1`)` `>valorNat(`d_2`)`
  else
  `valorString(`d_1`)` `>valorString(`d_2`)`
  fi **fi** }
  Complejidad: $\mathcal{O}(\text{cmp}(string))$
  Descripción: Se asume que los valores string son mayores a los valores nat para tener un orden total y poder usarlo como claves en un diccionario

2.2. Representación

`dato` se representa con `vec`

donde `vec` es `tupla(`*esNat*`: bool,` *valorNat*`: nat,` *valorString*`: string)`

2.2.1. Invariante de representación

Rep : *estr* → boolean

(∀ *e*: *estr*) Rep(*e*) ≡ true

2.2.2. Función de abstracción

Abs : *e*: *estr* → *dato* {Rep(*e*)}

(∀ *e*: *estr*) Abs(*e*) $=_{\text{obs}}$ *d*: *dato* ⟺

- $e.esNat \iff Nat?(d) \land$
- $(e.esNat \implies e.valorNat =_{\text{obs}} valorNat(d)) \land$
- $(\neg e.esNat \implies e.valorString =_{\text{obs}} valorString(d))$

2.3. Algoritmos

iDatoNat(**in** *n*: *nat*) → *res*: **dato**	
$res.esNat \leftarrow true$	// $\mathcal{O}(1)$
$res.valorNat \leftarrow n$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	
Justificación: $\mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$	

iDatoString(**in** *str*: *string*) → *res*: **dato**	
$res.esNat \leftarrow false$	// $\mathcal{O}(1)$
$res.valorString \leftarrow str$	// $\mathcal{O}(copy(string))$
Complejidad: $\mathcal{O}(copy(string))$	
Justificación: $\mathcal{O}(1) + \mathcal{O}(copy(string)) = \mathcal{O}(copy(string))$	

iIsNat(**in** *d*: *dato*) → *res*: **bool**	
$res \leftarrow d.esNat$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	

iIsString(**in** d: $dato$) $\rightarrow$ res: **bool**	
$res \leftarrow !\, d.esNat$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	

iGetNat(**in** d: $dato$) $\rightarrow$ res: **nat**	
$res \leftarrow d.valorNat$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	

iGetString(**in** d: $dato$) $\rightarrow$ res: **string**	
$res \leftarrow d.valorString$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	

$\bullet == \bullet$(**in** d_1 : $dato$, **in** d_2 : $dato$) $\rightarrow$ res: **bool**	
if d_1.esNat **then**	
$res \leftarrow d_1.valorNat == d_2.valorNat$	// $\mathcal{O}(1)$
else	
$res \leftarrow d_1.valorString == d_2.valorString$	// $\mathcal{O}(cmp(string))$
end if	
Complejidad: $\mathcal{O}(\text{cmp}(string))$	
Justificación: $\mathcal{O}(1) + \mathcal{O}(cmp(string)) = \mathcal{O}(cmp(string))$	

$\bullet > \bullet$(**in** d_1 : $dato$, **in** d_2 : $dato$) $\rightarrow$ res: **bool**	
if d_1.esNat **then**	
$res \leftarrow d_1.valorNat > d_2.valorNat$	// $\mathcal{O}(1)$
else	
$res \leftarrow d_1.valorString > d_2.valorString$	// $\mathcal{O}(cmp(string))$
end if	
Complejidad: $\mathcal{O}(cmp(string))$	
Justificación: $\mathcal{O}(1) + \mathcal{O}(cmp(string)) = \mathcal{O}(cmp(string))$	

2.4. Servicios usados

3. Tabla

3.1. Interfaz

usa:

se explica con: Tabla

géneros: tabla

operaciones:

- n es la cantidad de registros presentes en la tabla
- L es el máximo largo de un dato string
- La cantidad de campos se asume acotada por constante
- Los nombres de tablas y campos se asumen acotados por constante

NuevaTabla(**in** *nombre*: *string*, **in** *claves*: *conj*(*campos*), **in** *columnas*: *registro*) → *res*: **tabla**
Pre ≡ {¬∅?(*claves*) ∧ *claves* ⊆ *campos*(*columnas*)}
Post ≡ {*res* =$_{\text{obs}}$ *nuevaTabla*(*nombre*, *claves*, *columnas*)}
Complejidad: $\mathcal{O}(1)$
Aliasing: Los parámetros pasados no deben ser modificados luego

AgregarRegistro(**in/out** *t*: *tabla*, **in** r: *registro*)
Pre ≡ {*t* =$_{\text{obs}}$ t_0 ∧ *campos*(*r*) =$_{\text{obs}}$ *campos*(*t*) ∧ *puedoInsertar?*(*r*, *t*)}
Post ≡ {*t* =$_{\text{obs}}$ *agregarRegistro*(*r*, t_0)}
Complejidad: $\mathcal{O}(L + in)$
Descripción: *in* es $\mathcal{O}(log(n))$ si hay un indice sobre un campo de tipo NAT, $\mathcal{O}(1)$ sino.

BorrarRegistro(**in/out** *t*: *tabla*, **in** *c*: *campo*, **in** *d*: *dato*) → *res*: *conj*(*registro*)
Pre ≡ {*t* =$_{\text{obs}}$ t_0 ∧ *c* ∈ *claves*(*t*) ∧$_L$ *tipo?*(*d*) =$_{\text{obs}}$ *tipoCampo*(*c*, *t*)}
Post ≡ {*t* =$_{\text{obs}}$ *borrarRegistro*(*crit*, t_0)}
Complejidad: $\mathcal{O}(L + log(n))$ si el campo de borrado es un índice, $\mathcal{O}(L * n)$ sino

Indexar(**in/out** *t*: *tabla*, **in** c: *campo*)
Pre ≡ {*t* =$_{\text{obs}}$ t_0 ∧ *puedeIndexar*(*c*, *t*)}
Post ≡ {*t* =$_{\text{obs}}$ *indexar*(*c*, t_0)}
Complejidad: $\mathcal{O}(n * (L + log(n)))$

Nombre(**in** *t*: *tabla*) → *res*: **string**
Pre ≡ {`true`}
Post ≡ {*res* =$_{\text{obs}}$ *nombre*(*t*)}
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar la string retornada

EsClave(**in** *t*: *tabla* **in** *c*: *campo*) → *res*: **bool**
Pre ≡ {*c* ∈ *campos*(*t*)}
Post ≡ {*res* =$_{\text{obs}}$ *c* ∈ *claves*(*t*}
Complejidad: $\mathcal{O}(1)$
Descripción: La complejidad es $\mathcal{O}(|campos| * cmp(string))$, pero ambos entán acotados

EsIndice(**in** *t*: *tabla* **in** *c*: *campo*) → *res*: **bool**
Pre ≡ {*c* ∈ *campos*(*t*)}
Post ≡ {*res* =$_{\text{obs}}$ *c* ∈ *indices*(*t*}
Complejidad: $\mathcal{O}(1)$

REGISTROS(**in** t : *tabla*) $\rightarrow$ *res*: **itConj(registro)**
Pre $\equiv$ {`true`}
Post $\equiv$ $\{alias(esPermutacion?(SecuSuby(res), toList(registros(t)))) \wedge vacia?(Anteriores(res))\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar el conjunto retornado

CAMPOS(**in** t : *tabla*) $\rightarrow$ *res*: **registro**
Pre $\equiv$ {`true`}
Post $\equiv$ $\{claves(res) =_{\text{obs}} campos(t) \wedge_L$
$(\forall c : campo)c \in campos(t) \Rightarrow_L tipo?(obtener(c, res)) =_{\text{obs}} tipoCampo(c, t)\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar el conjunto retornado

TIPOCAMPO(**in** t : *tabla*, **in** c : *campo*) $\rightarrow$ *res*: **bool**
Pre $\equiv$ $\{c \in campos(t)\}$
Post $\equiv$ $\{res =_{\text{obs}} tipoCampo(c, t)\}$
Complejidad: $\mathcal{O}(1)$

ACCESOS(**in** t : *tabla*) $\rightarrow$ *res*: **nat**
Pre $\equiv$ {`true`}
Post $\equiv$ $\{res =_{\text{obs}} cantidadDeAccesos(t)\}$
Complejidad: $\mathcal{O}(1)$

MAXNAT(**in** t : *tabla*) $\rightarrow$ *res*: **nat**
Pre $\equiv$ $\{\neg\emptyset?(registros(t)) \wedge (\exists c : campo)c \in indices(t) \wedge_L tipoCampo(c, t)\}$
Post $\equiv$ $\{(\exists c : campo)c \in indices(t) \wedge_L tipoCampo(c, t) \wedge res =_{\text{obs}} valorNat(maximo(c, t))\}$
Complejidad: $\mathcal{O}(1)$

MINNAT(**in** t : *tabla*) $\rightarrow$ *res*: **nat**
Pre $\equiv$ $\{\neg\emptyset?(registros(t)) \wedge (\exists c : campo)c \in indices(t) \wedge_L tipoCampo(c, t)\}$
Post $\equiv$ $\{(\exists c : campo)c \in indices(t) \wedge_L tipoCampo(c, t) \wedge res =_{\text{obs}} valorNat(minimo(c, t))\}$
Complejidad: $\mathcal{O}(1)$

MAXSTRING(**in** t : *tabla*) $\rightarrow$ *res*: **string**
Pre $\equiv$ $\{\neg\emptyset?(registros(t)) \wedge (\exists c : campo)c \in indices(t) \wedge_L \neg tipoCampo(c, t)\}$
Post $\equiv$ $\{(\exists c : campo)c \in indices(t) \wedge_L \neg tipoCampo(c, t) \wedge res =_{\text{obs}} valorString(maximo(c, t))\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar el resultado

MINSTRING(**in** t : *tabla*) $\rightarrow$ *res*: **string**
Pre $\equiv$ $\{\neg\emptyset?(registros(t)) \wedge (\exists c : campo)c \in indices(t) \wedge_L \neg tipoCampo(c, t)\}$
Post $\equiv$ $\{(\exists c : campo)c \in indices(t) \wedge_L \neg tipoCampo(c, t) \wedge res =_{\text{obs}} valorString(minimo(c, t))\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar el resultado

BUSCAR(**in** t : *tabla*, **in** r : *registro*) $\rightarrow$ *res*: **conj(registro)**
Pre $\equiv$ $\{claves(r) \subseteq campos(t) \wedge_L (\forall c : campo)c \in claves(r) \Rightarrow_L tipo?(obtener(c, r)) =_{\text{obs}} tipoCampo(c, t)\}$
Post $\equiv$ $\{res =_{\text{obs}} coincidencias(r, registros(t))\}$
Complejidad: $\mathcal{O}(L)$ si hay un campo clave e índice string,
$\mathcal{O}(L + log(n))$ si hay un campo clave e índice nat,
$\mathcal{O}(L * n)$ si no
Aliasing: No se debe modificar los resultados

CANTIDADREGISTROS(**in** t : *tabla*) $\rightarrow$ *res*: **nat**
Pre $\equiv$ {`true`}
Post $\equiv$ $\{res =_{\text{obs}} \#registros(t)\}$
Complejidad: $\mathcal{O}(1)$

3.2. Representación

La estructura mantine un conjunto de registros y, si se encuentra indexada, un árbol con el índice string basado en un trie y otro con el índice nat basado en un avl.

Ademas se guarda su nombre, el conjunto de campos, el conjunto de claves, y la cantidad de accesos que tuvo.

`tabla` **se representa con** `vec`

donde `vec` es `tupla(`*registros*`: regs,` *hayIndiceNat*`: bool,` *indiceNat*`: campo,` *indicesNat*`: treeNat,`
hayIndiceString`: bool,` *indiceString*`: campo,` *indicesString*`: treeString,`
nombre`: nombreTabla,` *campos*`: registro,` *claves*`: conj(`*campo*`),` *accesos*`: nat)`

donde `regs` es `conj(`*registro*`)`

donde `treeNat` es `diccLog(`*nat*`, conj(itConj(`*registro*`)))`

donde `treeString` es `diccTrie(conj(itConj(`*registro*`)))`

donde `campo` es `string`

donde `nombreTabla` es `string`

donde `registro` es `diccLog(`*campo*`,` *dato*`)`

3.2.1. Invariante de representación

Rep : $estr \to$ boolean

$(\forall\ e\colon estr)$ Rep$(e) \equiv$ true $\iff$

- Hay claves y están incluidas en campos
  $\neg\emptyset?(e.claves) \wedge e.claves \subseteq claves(e.campos)$
- Los registros tienen los mismos campos y del mismo tipo
  $(\forall r : registro)(r \in e.registros \ \Rightarrow\ claves(r) = claves(e.campos) \wedge_L$
  $(\forall c : campo)(c \in claves(r) \ \Rightarrow\ _L\ tipo?(obtener(c, r)) = tipo?(obtener(c, e.campos))))$
- Los campos clave no tienen repetidos
  $(\forall c : campo)(\forall r_1, r_2 : registro)$
  $c \in e.claves \wedge r_1 \in e.registros \wedge r_1 \in e.registros \wedge_L obtener(c, r_1) =_{\text{obs}} obtener(c, r_2) \ \Rightarrow\ r_1 =_{\text{obs}} r_2$
- Si no hay indice de tipo nat, indicesNat está vacio
  $\neg e.hayIndiceNat \ \Rightarrow\ \emptyset?(e.indicesNat)$
- Si hay indice de tipo nat, indiceNat es un campo válido e indicesNat tiene los registros referenciados por el índice
  $e.hayIndiceNat \ \Rightarrow\ indiceNat \in claves(e.campos) \wedge_L Nat?(obtener(indiceNat, e.campos)) \wedge_L$
  $(\forall r : registro)(r \in e.registros \ \Rightarrow\ def?(obtener(indiceNat, r), e.indicesNat) \wedge_L$
  $(\exists it : itConj(registro))\ it \in obtener(obtener(indiceNat, r), e.indicesNat) \wedge$
  $haySiguiente?(it) \wedge_L siguiente(it) =_{\text{obs}} r) \wedge$
  $(\forall n : nat)(\forall it : itConj(registro))($
  $def?(e.indicesNat, n) \wedge_L it \in obtener(e.indicesNat, n) \ \Rightarrow$
  $haySiguiente?(it) \wedge_L siguiente(it) \in e.registros)$
- Si no hay indice de tipo string, indicesString está vacio
  $\neg e.hayIndiceString \ \Rightarrow\ \emptyset?(e.indicesString)$
- Si hay indice de tipo string, indiceString es un campo válido e indicesString tiene los registros referenciados por el índice
  $e.hayIndiceString \ \Rightarrow\ indiceString \in claves(e.campos) \wedge_L String?(obtener(indiceString, e.campos)) \wedge_L$
  $(\forall r : registro)(r \in e.registros \ \Rightarrow\ def?(obtener(indiceString, r), e.indicesString) \wedge_L$
  $(\exists it : itConj(registro))\ it \in obtener(obtener(indiceString, r), e.indicesString) \wedge$
  $haySiguiente?(it) \wedge_L siguiente(it) =_{\text{obs}} r) \wedge$

$$(\forall s : string)(\forall it : itConj(registro))(\\ def?(e.indicesString, s) \wedge_L it \in obtener(e.indicesString, s) \Rightarrow \\ hayS iguiente?(it) \wedge_L siguiente(it) \in e.registros)$$

- Hay por lo menos tantos accesos como registros insertados
  $e.accesos \geq \#e.registros$

3.2.2. Función de abstracción

Abs : $e : estr \to tabla$ {Rep(e)}

($\forall\, e : estr$) Abs(e) $=_{\text{obs}}$ $t : tabla \iff$

- $e.nombre =_{\text{obs}} nombre(t) \wedge$
- $e.claves =_{\text{obs}} claves(t) \wedge$
- $e.registros =_{\text{obs}} r \in registros(t) \wedge$
- $e.accesos =_{\text{obs}} cantidadDeAccesos(t) \wedge$
- (**if** $e.hayIndiceNat$ **then** $Ag(indiceNat, \emptyset)$ **else** $\emptyset$ **fi**) $\cup$
  (**if** $e.hayIndiceString$ **then** $Ag(indiceString, \emptyset)$ **else** $\emptyset$ **fi**) $=_{\text{obs}} indices(t) \wedge$
- $claves(e.campos) =_{\text{obs}} campos(t) \wedge_L$
- $(\forall c : campo)\ (c \in claves(e.campos) \Rightarrow_L tipo?(obtener(c, e.campos)) =_{\text{obs}} tipoCampo(c, t))$

3.3. Algoritmos

iNuevaTabla(**in** *nombre* : *string*, **in** *claves* : *conj*(*campos*), **in** *columnas* : *registro*) $\to$ *res*: **tabla**	
$res.registros \leftarrow nuevoDiccLog()$	// $\mathcal{O}(1)$
$res.hayIndiceNat \leftarrow false$	// $\mathcal{O}(1)$
$res.hayIndiceString \leftarrow false$	// $\mathcal{O}(1)$
$res.nombre \leftarrow nombre$	// $\mathcal{O}(1)$
$res.campos \leftarrow columnas$	// $\mathcal{O}(1)$
$res.claves \leftarrow claves$	// $\mathcal{O}(1)$
$res.accesos \leftarrow 0$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	
Justificación: 7 * $\mathcal{O}(1) = \mathcal{O}(1)$	

iAgregarRegistro(**in/out** t : $tabla$, **in** r : $registro$)

```
var it : itConj(registro)
it ← agregarRapido(t.registros, r)                                   // O(copy(registro)) = O(L)
if t.hayIndiceNat then
    var k : nat
    k ← getNat(significado(r, t.indiceNat))                          // O(1)
    if !definido?(t.indicesNat, k) then                              // O(log(n) * cmp(nat)) = O(log(n))
        definir(t.indicesNat, k, vacio())                            // O(log(n) * cmp(nat)) = O(log(n))
    end if
    agregarRapido(significado(t.indicesNat, k, it))                  // O(log(n) * cmp(nat) + copy(nat)) = O(log(n))
end if
if t.hayIndiceString then
    var k : string
    k ← getString(significado(r, t.indiceString))                    // O(1)
    if !definido?(t.indicesString, k) then                           // O(L)
        definir(t.indicesString, k, vacio())                         // O(L)
    end if
    agregarRapido(significado(t.indicesString, k, it))               // O(L + copy(string)) = O(L)
end if

t.accesos ← t.accesos + 1                                            // O(1)
```

Complejidad: $\mathcal{O}(L + log(n))$ si hay un índice nat, $\mathcal{O}(L)$ sino

Justificación: Si hay índice nat: $\mathcal{O}(L) + \mathcal{O}(1) + \mathcal{O}(log(n)) + \mathcal{O}(1) + \mathcal{O}(L) + \mathcal{O}(1) = \mathcal{O}(L + log(n))$
Sino: $\mathcal{O}(L) + \mathcal{O}(1) + \mathcal{O}(L) + \mathcal{O}(1) = \mathcal{O}(L)$

iBorrarRegistro(**in/out** t: *tabla*, **in** c: *campo*, **in** d: *dato*) $\rightarrow$ *res*: $conj(registro)$

```
var esta : bool
var its : conj(itConj(registro))
var it : itConj(registro)
res ← vacio()                                                          // O(1)
esta ← false                                                           // O(1)

// Por requerimientos de la especificación, c siempre es clave
// Por lo tanto, se borra a lo sumo un registro

if t.hayIndiceNat && t.indiceNat == c then                             // O(1)
    if definido(t.indicesNat, getNat(d)) then                          // O(log(n) * cmp(nat)) = O(log(n))
        // Como c es clave, hay un único iterador en el conjunto
        its ← obtener(t.indicesNat, getNat(d))                         // O(log(n) * cmp(nat)) = O(log(n))
        it ← siguiente(crearIt(its))                                   // O(1)
        esta ← true                                                    // O(1)
    end if
else if t.hayIndiceString && t.indiceString == c then                  // O(1)
    if definido(t.indicesString, getString(d)) then                    // O(L)
        // Como c es clave, hay un único iterador en el conjunto
        its ← obtener(t.indicesString, getString(d))                   // O(L)
        it ← siguiente(crearIt(its))                                   // O(1)
        esta ← true                                                    // O(1)
    end if
else
    it ← crearIt(t.registros)                                          // O(1)
    while haySiguiente(it) && significado(siguiente(it), c) != d do    // O(1)
        // El loop se repite a lo sumo n veces
        avanzar(it)                                                    // O(1)
    end while
    if haySiguiente(it) then                                           // O(1)
        esta ← true                                                    // O(1)
    end if
end if

if esta then
    var r : registro
    // Agregamos una copia del registro borrado a res
    r ← copiar(siguiente(it))                                          // O(L)
    agregarRapido(res, r)                                              // O(1)

    eliminarSiguiente(it)                                              // O(1)
    if t.hayIndiceNat then                                             // O(1)
        borrar(t.indicesNat, significado(r, t.indiceNat))              // O(log(n) * cmp(nat)) = O(log(n))
    end if
    if t.hayIndiceString then                                          // O(1)
        borrar(t.indicesString, significado(r, t.indiceString))        // O(L)
    end if

    t.accesos ← t.accesos + 1                                          // O(1)
end if
```

Complejidad: $\mathcal{O}(L + log(n))$ si c es índice, $\mathcal{O}(L + n)$ sino

Justificación: Si c es indiceNat: 3 * $\mathcal{O}(1)$ + 2 * $\mathcal{O}(log(n))$ + $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + 3 * $\mathcal{O}(1)$ + $\mathcal{O}(log(n))$ + $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$ = $\mathcal{O}(L + log(n))$

Si c es indiceString: 4 * $\mathcal{O}(1)$ + 2 * $\mathcal{O}(L)$ + $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + 3 * $\mathcal{O}(1)$ + $\mathcal{O}(log(n))$ + $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$ = $\mathcal{O}(L + log(n))$

Si c no es indice: 5 * $\mathcal{O}(1)$ + n * (2 * $\mathcal{O}(1)$) + 2 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + 3 * $\mathcal{O}(1)$ + $\mathcal{O}(log(n))$ + $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$ = $\mathcal{O}(L + n)$

iIndexar(**in/out** t : *tabla*, **in** c : *campo*)

var it : itConj(registro)
$it \leftarrow crearIt(t.registros)$ // $\mathcal{O}(1)$
if tipoCampo(t,c) **then** // $\mathcal{O}(1)$
  $t.hayIndiceNat \leftarrow true$ // $\mathcal{O}(1)$
  $t.indiceNat \leftarrow c$ // $\mathcal{O}(1)$
  $t.indicesNat \leftarrow vacio()$ // $\mathcal{O}(1)$
  while $haySiguiente(it)$ **do** // $\mathcal{O}(1)$
    // El loop se repite a lo sumo n veces
    var $k : nat$
    $k \leftarrow getNat(significado(siguiente(it), c))$ // $\mathcal{O}(1)$
    if $!definido?(t.indicesNat, k)$ **then** // $\mathcal{O}(log(n) * cmp(nat)) = \mathcal{O}(log(n))$
      $definir(t.indicesNat, k, vacio())$ // $\mathcal{O}(log(n) * cmp(nat)) = \mathcal{O}(log(n))$
    end if
    $agregarRapido(significado(t.indicesNat, k, copy(it)))$ // $\mathcal{O}(log(n) * cmp(nat) + copy(nat)) = \mathcal{O}(log(n))$
    $avanzar(it)$ // $\mathcal{O}(1)$
  end while
else
  $t.hayIndiceString \leftarrow true$ // $\mathcal{O}(1)$
  $t.indiceString \leftarrow c$ // $\mathcal{O}(1)$
  $t.indicesString \leftarrow vacio()$ // $\mathcal{O}(1)$
  while $haySiguiente(it)$ **do** // $\mathcal{O}(1)$
    // El loop se repite a lo sumo n veces
    var $k : string$
    $k \leftarrow getString(significado(siguiente(it), c))$ // $\mathcal{O}(1)$
    if $!definido?(t.indicesString, k)$ **then** // $\mathcal{O}(L)$
      $definir(t.indicesString, k, vacio())$ // $\mathcal{O}(L)$
    end if
    $agregarRapido(significado(t.indicesString, k, copy(it)))$ // $\mathcal{O}(L + copy(string)) = \mathcal{O}(L)$
    $avanzar(it)$ // $\mathcal{O}(1)$
  end while
end if

Complejidad: $\mathcal{O}(n * (L + log(n)))$
Justificación: Si el campo es nat: 5 * $\mathcal{O}(1)$ + n * ($\mathcal{O}(1)$ + $\mathcal{O}(log(n))$ + $\mathcal{O}(1)$) = $\mathcal{O}(n * log(n))$
Sino: 5 * $\mathcal{O}(1)$ + n * ($\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$) = $\mathcal{O}(n * L)$

---

iNombre(**in** t : *tabla*) $\rightarrow$ *res*: **string**

$res \leftarrow t.nombre$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iEsClave(**in** t : *tabla*, **in** c : *campo*) $\rightarrow$ *res*: **bool**

$res \leftarrow pertenece?(t.claves, c)$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iEsIndice(**in** t : *tabla*, **in** c : *campo*) $\rightarrow$ *res*: **bool**

if tipoCampo(t,c) **then** // $\mathcal{O}(1)$
  $res \leftarrow c == t.indiceNat$ // $\mathcal{O}(1)$
else
  $res \leftarrow c == t.indiceString$ // $\mathcal{O}(1)$
end if

Complejidad: $\mathcal{O}(1)$
Justificación: 3 * $\mathcal{O}(1) = \mathcal{O}(1)$

---

iRegistros(**in** t : *tabla*) $\rightarrow$ *res*: **itConj(registro)**

$res \leftarrow crearIt(t.registros)$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iCampos(**in** t : *tabla*) $\rightarrow$ *res*: **registro**

$res \leftarrow t.campos$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iTipoCampo(**in** t : *tabla*, **in** c : *campo*) $\rightarrow$ *res*: **registro**

$res \leftarrow isNat(significado(t.campos, c))$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iAccesos(**in** t : *tabla*) $\rightarrow$ *res*: **nat**

$res \leftarrow t.accesos$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

maxNat(**in** t : *tabla*) $\rightarrow$ *res*: **nat**

$res \leftarrow maximo(t.indicesNat)_0$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

minNat(**in** t : *tabla*) $\rightarrow$ *res*: **nat**

$res \leftarrow minimo(t.indicesNat)_0$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

maxString(**in** t : *tabla*) $\rightarrow$ *res*: **string**

$res \leftarrow maximo(t.indicesString)_0$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

minString(**in** t : *tabla*) $\rightarrow$ *res*: **string**

$res \leftarrow minimo(t.indicesString)_0$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iBuscar(**in/out** t : *tabla*, **in** c : *campo*, **in** d : *dato*) $\rightarrow$ *res*: $conj(registro)$

```
var it : itConj(registro)
var c : campo
var d : dato
res ← vacio()                                                                    // O(1)

if t.hayIndiceString && def?(r, t.indiceString) && pertenece?(t.claves, t.indiceString) then   // O(1)
    // Hay un campo clave e índice string
    c ← t.indiceString                                                           // O(1)
    d ← obtener(r, c)                                                            // O(1)

    if definido(t.indicesString, getString(d)) then                              // O(L)
        // Como c es clave, hay un único registro con este valor
        it ← siguiente(obtener(t.indicesString, getString(d)))                   // O(L)
        agregarRapido(res, siguiente(it))                  // O(copy(registro)) = O(L)
    end if
else if t.hayIndiceNat && def?(r, t.indiceNat) && pertenece?(t.claves, t.indiceNat) then   // O(1)
    // Hay un campo clave e índice nat
    c ← t.indiceNat                                                              // O(1)
    d ← obtener(r, c)                                                            // O(1)

    if definido(t.indicesNat, getNat(d)) then          // O(log(n) * cmp(nat)) = O(log(n))
        // Como c es clave, hay un único registro con este valor
        it ← siguiente(obtener(t.indicesNat, getNat(d)))   // O(log(n) * cmp(nat)) = O(log(n))
        agregarRapido(res, siguiente(it))                  // O(copy(registro)) = O(L)
    end if
else
    // Hacemos una búsqueda lineal sobre los registros
    it ← crearIt(t.registros)                                                    // O(1)
    while haySiguiente(it) do                                                    // O(1)
        // El loop se repite n veces
        var itCs : itDiccLog(campo, dato)
        var r_tabla : registro
        var todoIgual : bool
        r_tabla ← obtener(siguiente(it))                                         // O(1)
        itCs : crearIt(r)                                                        // O(1)
        todoIgual ← true                                                         // O(1)
        while hayMas?(itCs) do                                                   // O(1)
            // El loop se repite a lo sumo |campos(t)| veces, acotado por constante
            if obtener(actual(itCs).clave, r_tabla) ≠ obtener(actual(itCs).significado then // O(cmp(dato)) = O(L)
                todoIgual ← false                                                // O(1)
            end if
            avanzar(itCs)                                                        // O(1)
        end while
        if todoIgual then                                                        // O(1)
            agregarRapido(res, r_tabla)                    // O(copy(registro)) = O(L)
        end if
        avanzar(it)                                                              // O(1)
    end while
end if
```

Complejidad: $\mathcal{O}(L + log(n))$ si c es índice, $\mathcal{O}(L * n)$ sino

Justificación: Si hay un campo en r que es clave e indiceNat: 4 * $\mathcal{O}(1)$ + 2 * $\mathcal{O}(log(n))$ + $\mathcal{O}(L) = \mathcal{O}(L + log(n))$
Si hay un campo en r que es clave e indiceString: 3 * $\mathcal{O}(1)$ + 3 * $\mathcal{O}(L) = \mathcal{O}(L)$
Si no: 4 * $\mathcal{O}(1)$ + n * (4 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$ + $\mathcal{O}(L)$) = $\mathcal{O}(L * n)$

iCantidadRegistros(**in** t : *tabla*) $\rightarrow$ *res*: **nat**	
$res \leftarrow cardinal(t.registros)$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	

3.4. Servicios usados

Se utilizan las operaciones exportadas por el módulo Dato con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo DiccLog con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo DiccTrie con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo Conjunto Lineal (α) con sus complejidades declaradas.

4. BaseDeDatos

4.1. Interfaz

usa:

se explica con: BaseDeDatos

géneros: db

operaciones:

- T es la cantidad de tablas en la db
- n y m son la cantidad de registros presentes en las tablas sobre las que se opera
- L es el máximo largo de un dato string
- La cantidad de campos se asume acotada por constante
- Los nombres de tablas y campos se asumen acotados por constante

NuevaDB() $\rightarrow$ *res*: **base**
Post $\equiv \{res =_{\text{obs}} nuevaDB()\}$
Complejidad: $\mathcal{O}(1)$

AgregarTabla(**in/out** db: $base$, **in** t: $tabla$)
Pre $\equiv \{db =_{\text{obs}} db_0 \wedge \emptyset?(registros(t)) \wedge \neg(nombre(t) \in tablas(db))\}$
Post $\equiv \{db =_{\text{obs}} agregarTabla(t, db_0)\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar la tabla después de insertada

InsertarEntrada(**in/out** db: $base$, **in** nt: $nombreTabla$, **in** r: $registro$)
Pre $\equiv \{db =_{\text{obs}} db_0 \wedge nt \in tablas(db) \wedge_L puedoInsertar?(r, dameTabla(nt, db))\}$
Post $\equiv \{db =_{\text{obs}} insertarEntrada(t, db_0)\}$
Complejidad: $\mathcal{O}(T * L + in)$
Descripción: in es $\mathcal{O}(log(n))$ si hay un índice sobre un campo de tipo NAT, $\mathcal{O}(1)$ sino

Borrar(**in/out** db: $base$, **in** nt: $nombreTabla$, **in** c: $campo$, **in** d: $dato$)
Pre $\equiv \{db =_{\text{obs}} db_0 \wedge nt \in tablas(db) \wedge c \in claves(dameTabla(nt, db)) \wedge_L$
$tipo?(d) =_{\text{obs}} tipoCampo(c, dameTabla(nt, db))\}$
Post $\equiv \{db =_{\text{obs}} borrar(definir(c, d, vacio), db_0)\}$
Complejidad: $\mathcal{O}(T * L + log(n))$ si el campo de borrado es un índice, $\mathcal{O}(L * (T + n))$ sino

GenerarVistaJoin(**in/out** db: $base$, **in** nt_1: $nombreTabla$,
in nt_2: $nombreTabla$, **in** c: $campo$) $\rightarrow$ *res*: **itConj(registro)**
Pre $\equiv \{db =_{\text{obs}} db_0 \wedge nt_2 \in tablas(db) \wedge nt_1 \in tablas(db) \wedge nt_1 \neq nt_2 \wedge_L$
$c \in claves(dameTabla(nt_1, db)) \wedge c \in claves(dameTabla(nt_2, db)) \wedge_L$
$tipoCampo(c, dameTabla(nt_1, db)) =_{\text{obs}} tipoCampo(c, dameTabla(nt_2, db)) \wedge$
$\neg hayJoin?(nt_1, nt_2, db)\}$
Post $\equiv \{db =_{\text{obs}} generarVistaJoin(nt_1, nt_2, c, db_0) \wedge_L$
$alias(esPermutacion?(SecuSuby(res), vistaJoin(nt_1, nt_2, db))) \wedge vacia?(Anteriores(res))\}$
Complejidad: $\mathcal{O}(min(n, m) * (L + log(n + m)))$ si el campo es indice de ambas tablas,
$\mathcal{O}(n * (L + log(n + m)))$ si el campo es sólo índice de t_2,
$\mathcal{O}(m * (L + log(n + m)))$ si el campo es sólo índice de t_1 y
$\mathcal{O}(m * (n * L + log(n)))$ si el campo no es un índice,
donde $n = |registros(t_1)|$ y $m = |registros(t_2)|$

HayTabla(**in** db: $base$, **in** nt: $nombreTabla$) $\rightarrow$ *res*: *bool*
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} nt \in tablas(db)\}$
Complejidad: $\mathcal{O}(1)$

TABLA(**in** db: $base$, **in** nt: $nombreTabla$) $\rightarrow$ res: $tabla$
Pre $\equiv$ $\{nt \in tablas(db)\}$
Post $\equiv$ $\{res =_{\text{obs}} dameTabla(nt, db)\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar la tabla devuelta

TABLAS(**in** db: $base$) $\rightarrow$ res: $itConj(tabla)$
Pre $\equiv$ $\{\texttt{true}\}$
Post $\equiv$ $\{alias(esPermutacion?(SecuSuby(res), getTablasConj(tablas(db), db))) \wedge vacia?(Anteriores(res))\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar las tablas
Descripción: El iterador devuelto tiene $siguiente()$ en $\mathcal{O}(1)$

TABLAMAXIMA(**in** db: $base$) $\rightarrow$ res: $tabla$
Pre $\equiv$ $\{\#tablas(db) > 0\}$
Post $\equiv$ $\{res =_{\text{obs}} dameTabla(tablaMaxima(db), db)\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: No se debe modificar la tabla devuelta

HAYJOIN(**in** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$) $\rightarrow$ res: $bool$
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} hayJoin?(nt_1, nt_2, db)\}$
Complejidad: $\mathcal{O}(1)$

CAMPOJOIN(**in** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$) $\rightarrow$ res: $campo$
Pre $\equiv$ $\{hayJoin?(nt_1, nt_2, db)\}$
Post $\equiv$ $\{res =_{\text{obs}} campoJoin(nt_1, nt_2, db)\}$
Complejidad: $\mathcal{O}(1)$

BORRARJOIN(**in/out** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$)
Pre $\equiv$ $\{db =_{\text{obs}} db_0 \wedge hayJoin?(nt_1, nt_2, db)\}$
Post $\equiv$ $\{db =_{\text{obs}} borrar(nt_1, nt_2, db_0)\}$
Complejidad: $\mathcal{O}(1)$

VISTAJOIN(**in** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$) $\rightarrow$ res: $itConj(registro)$
Pre $\equiv$ $\{hayJoin?(nt_1, nt_2, db)\}$
Post $\equiv$ $\{alias(esPermutacion?(SecuSuby(res), vistaJoin(nt_1, nt_2, db))) \wedge vacia?(Anteriores(res))\}$
Complejidad: $\mathcal{O}(1)$ si $R = 0$,
$\mathcal{O}(R * (L + log(n * m)))$ si $R > 0$ y ambas tablas tienen índice en el campo del join y
$\mathcal{O}(R * L * (n + m))$ sino
R es la cantidad de modificaciónes sobre las tablas desde la generación o última visualización del join
Aliasing: No se debe modificar los resultados

BUSCAR(**in/out** db: $base$, **in** nt: $nombreTabla$, **in** r: $registro$) $\rightarrow$ res: **conj(registro)**
Pre $\equiv$ $\{nt \in tablas(db) \wedge_L (\exists t : tabla)\ (t =_{\text{obs}} dameTabla(nt, db) \wedge$
$claves(r) \subseteq campos(t) \wedge_L (\forall c : campo)c \in claves(r)\ \Rightarrow\ _L tipo?(obtener(c, r)) =_{\text{obs}} tipoCampo(c, t))\}$
Post $\equiv$ $\{res =_{\text{obs}} buscar(r, nt, db)\}$
Complejidad: $\mathcal{O}(L)$ si hay un campo clave e índice string como criterio,
$\mathcal{O}(L + log(n))$ si hay un campo clave e índice nat,
$\mathcal{O}(L * n)$ si no,
donde $n = |registros(dameTabla(nt, db)|)$
Aliasing: No se debe modificar los resultados

4.1.1. Operaciones auxiliares de la interfaz

getTablasConj : conj(string) nts × base db $\longrightarrow$ conj(tabla) $\{nts \subseteq tablas(db)\}$

axiomas $\forall db : base, \forall nts : conj(string), \forall as : conj(\alpha)$

getTablasConj(nts,db) $\equiv$ **if** $\emptyset$?(nts) **then**
$\emptyset$
else
Ag(dameTabla(dameUno(nts),db), getTablasConj(sinUno(nts),db))
fi

4.2. Representación

La estructura mantine un conjunto de tablas y un índice sobre sus nombres basado en un trie.

Cada join se mantiene cacheado con un conjunto de los registros resultantes y un índices sobre el campo del join usando un avl o un trie según su tipo. También se mantiene una cola de modificaciónes sobre una lista enlazada que se procesa al pedir el resultado del join.

Por último, se mantiene un puntero a una de las tablas con mas modificaciones.

`db` se representa con `vec`

donde `vec` es `tupla(`$tablas$`: tablas,` $tablasTree$`: tablasTree,` $vistasJoin$`: vistas,` $tablaMaxima$`: maxima)`

donde `tablas` es `conj(`$tabla$`)`

donde `tablasTree` es `diccTrie(`$nombreTabla$`, itConj(`$tabla$`))`

donde `maxima` es `itConj(`$tabla$`)`

donde `vistas` es `diccTrie(`$nombreTabla$`, diccTrie(`$nombreTabla$`,` $vistaJoin$`))`

donde `vistaJoin` es `tupla(`$campo$`: campo,` $tipo$`: bool,` $buffer$`: buffer,`
$joins$`: joins,` $joinsNat$`: jNat,` $joinsString$`: jString)`

donde `joins` es `conj(`$registro$`)`

donde `joinsNat` es `diccLog(`$nat, itConj(registro)$`)`

donde `joinsString` es `diccTrie(`$itConj(registro)$`)`

donde `buffer` es `lista(`$operacionJoin$`)`

donde `operacionJoin` es `tupla(`$esInsercion$`: bool,` $enTablaB$`: bool,` reg`:` $registro$`)`

donde `nombreTabla` es `string`

donde `campo` es `string`

donde `registro` es `diccLog(`$campo$`,`$dato$`)`

4.2.1. Invariante de representación

Rep : $estr \rightarrow$ boolean

($\forall$ e: $estr$) Rep(e) $\equiv$ true $\iff$

- tablasTree tiene las tablas
  $\#e.tablas = \#claves(e.tablasTree) \wedge$
  $(\forall t : tabla)(t \in e.tablas \Rightarrow def?(nombre(t), e.tablasTree) \wedge_L$
  $hayS iguiente?(obtener(nombre(t), e.tablasTree)) \wedge_L$
  $Siguiente(obtener(nombre(t, e.tablasTree)) =_{\text{obs}} t))$
- vistasJoin tiene tablas válidas
  $(\forall nt : nombreTabla)(def?(nt, e.vistasJoin) \Rightarrow$
  $def?(nt, e.tablasTree) \wedge_L$

$(\forall nt' : nombreTabla)(def?(nt', obtener(nt, e.vistasJoin)) \Rightarrow$
$def?(nt', e.tablasTree))$

- El campo de las vistasJoin es válido

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin)) \Rightarrow_L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$
$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

$v.campo \in claves(t_a) \wedge$
$v.campo \in claves(t_b) \wedge_L$
$tipoCampo(v.campo, t_a) =_{\text{obs}} tipoCampo(v.campo, t_b) \wedge$
$v.tipo =_{\text{obs}} tipoCampo(v.campo, t_b)$

- Los registros de cada buffer tienen los campos de las tablas correspondientes

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin)) \Rightarrow_L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$
$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

$(\forall op : operacionJoin)\ op \in v.buffer \Rightarrow$
if $\neg op.enTablaB$ **then**
$campos(op.reg) =_{\text{obs}} campos(t_a) \wedge_L mismosTipos(op.reg, t_a)$
else
$campos(op.reg) =_{\text{obs}} campos(t_b) \wedge_L mismosTipos(op.reg, t_b)$
fi

- Los joins cacheados tienen los campos de ambas tablas

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin)) \Rightarrow_L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$
$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

$(\forall r : registro)\ r \in v.joins \Rightarrow ($
$campos(r) =_{\text{obs}} campos(t_a) \cup campos(t_b) \wedge$
$(\forall c : campo)\ c \in campos(r) \Rightarrow ($
$(c \in campos(t_a) \wedge_L tipoCampo(c, t_a) =_{\text{obs}} tipo?(obtener(c, r))) \vee$
$(c \in campos(t_b) \wedge_L tipoCampo(c, t_b) =_{\text{obs}} tipo?(obtener(c, r)))))$

- Los diccionarios de índices de los joins tienen todas las entradas

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin)) \Rightarrow_L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$
$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

if $v.tipo$ **then**
$\#v.joins =_{\text{obs}} \#claves(v.joinsNat) \wedge$
$(\forall r : registro)\ r \in v.joins \Rightarrow$
$def?(valorNat(obtener(v.campo, r)), v.joinsNat) \wedge_L$
$haySiguiente?(obtener(valorNat(obtener(v.campo, r)), v.joinsNat)) \wedge_L$
$siguiente(obtener(valorNat(obtener(v.campo, r)), v.joinsNat)) =_{\text{obs}} r$
else
$\#v.joins =_{\text{obs}} \#claves(v.joinsString) \wedge$
$(\forall r : registro)\ r \in v.joins \Rightarrow$
$def?(valorString(obtener(v.campo, r)), v.joinsString) \wedge_L$
$haySiguiente?(obtener(valorString(obtener(v.campo, r)), v.joinsString)) \wedge_L$
$siguiente(obtener(valorString(obtener(v.campo, r)), v.joinsString)) =_{\text{obs}} r$
fi

- No hay dos inserciones seguidas en un buffer (sin borrar antes)

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin)) \Rightarrow_L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$

$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

$(\forall op, op' : operacionJoin)\ (op \in v.buffer\ \wedge\ op' \in v.buffer\ \wedge_L$
$op.esInsercion\ \wedge\ op'.esInsercion\ \wedge$
$obtener(v.campo, op.reg) =_{\text{obs}} obtener(v.campo, op'.reg))\ \Rightarrow$
$(\exists op'' : operacionJoin)\ (op'' \in v.buffer\ \wedge_L$
$\neg op''.esInsercion\ \wedge$
$obtener(v.campo, op.reg) =_{\text{obs}} obtener(v.campo, op'.reg)\ \wedge$
$estaDespues?(op'', op, v.buffer)\ \wedge$
$estaDespues?(op', op'', v.buffer))$

- No hay inserciones de indices ya cacheados en un buffer (sin borrar antes)

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin))\ \Rightarrow\ _L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$
$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

$(\forall d : dato)\ (\forall op : operacionJoin)\ (op \in v.buffer\ \wedge\ op.esInsercion \wedge$
$datoIndexado?(d, v.joinsNat, v.joinsString) \wedge_L obtener(v.campo, op.reg) =_{\text{obs}} d)\ \Rightarrow$

$(\exists op' : operacionJoin)\ (op' \in v.buffer\ \wedge_L \neg op'.esInsercion\ \wedge$
$obtener(v.campo, op'.reg) =_{\text{obs}} d\ \wedge estaDespues?(op', op, v.buffer))$

- Todos los joins posibles están cacheados en vistasJoins y no hay modificaciones en el buffer, o la inserción está en el buffer y no se la borra luego

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin))\ \Rightarrow\ _L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$
$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

$(\forall r : registro) r \in combinarRegistros(v.campo, registros(t_a), registros(t_b))\ \Rightarrow\ _L($
$(r \in v.joins \wedge$
$\neg(\exists op' : operacionJoin)\ (op' \in v.buffer\ \wedge$
$obtener(v.campo, r) =_{\text{obs}} obtener(v.campo, op'.reg))) \vee$
$((\exists op : operacionJoin)\ op \in v.buffer\ \wedge\ op.esInsercion\ \wedge_L$
$coincidenTodos(r, campos(op.reg), op.reg) \wedge$
$\neg(\exists op' : operacionJoin)\ (op' \in v.buffer\ \wedge$
$obtener(v.campo, r) =_{\text{obs}} obtener(v.campo, op'.reg) \wedge$
$estaDespues?(op', op, v.buffer))))$

- vistasJoins tiene joins válidos o joins que seran eliminados

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin))\ \Rightarrow\ _L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$
$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

$(\forall r : registro)\ r \in v.joins\ \Rightarrow\ _L($
$r \in combinarRegistros(v.campo, registros(t_a), registros(t_b))$
$\vee$
$((\exists op : operacionJoin)\ op \in v.buffer\ \wedge_L$
$obtener(v.campo, r) =_{\text{obs}} obtener(v.campo, op.reg)))$

- Si hay inserciones de joins no válidos en el buffer, también está su eliminación

$(\forall nt, nt' : nombreTabla) def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin))\ \Rightarrow\ _L$
$(\exists v : vistaJoin)\ (\exists t_a, t_b : tabla)$
$v =_{\text{obs}} obtener(nt', obtener(nt, e.vistasJoin)) \wedge$
$t_a =_{\text{obs}} obtener(nt, e.tablasTree) \wedge t_b =_{\text{obs}} obtener(nt', e.tablasTree) \wedge$

$(\forall op : operacionJoin)\ (op \in v.buffer \wedge op.esInsercion \wedge$
$\emptyset?(coincidencias(op.reg,$

$$combinarRegistros(v.campo, registros(t_a), registros(t_b))))) \Rightarrow$$
$$((\exists op' : operacionJoin)\ op' \in v.buffer\ \wedge\ obtener(v.campo, op.reg) =_{\text{obs}} obtener(v.campo, op'.reg)\ \wedge\ estaDespues?(op', op, v.buffer))$$

- Si hay tablas, e.tablaMaxima apunta a una de las tablas con mas cantidadDeAccesos

$$\neg\emptyset?(e.tablas) \Rightarrow$$
$$hayProximo?(e.tablaMaxima) \wedge_L$$
$$proximo(e.tablaMaxima) \in e.tablas \wedge$$
$$(\forall t : tabla)\ t \in e.tablas \Rightarrow$$
$$cantidadDeAccesos(t) \leq cantidadDeAccesos(proximo(e.tablaMaxima))$$

4.2.2. Operaciones auxiliares del invariante de representación

estaDespues? : $\alpha\ a \times \alpha\ b \times \text{secu}(\alpha) \longrightarrow \text{bool}$ {esta?(a,s) ∧ esta?(b,s)}

datoIndexado? : d *dato* × dicc(nat α) *dn* × dicc(string β) *ds* ⟶ bool

axiomas ∀secu(α)s, ∀αa,b

estaDespues?(a,b,s) ≡ ¬(prim(s) = a) $\wedge_L$ (prim(s) = b $\vee_L$ estaDespues?(a,b,fin(s)))

datoIndexado?(d,dn,ds) ≡ **if** Nat?(d) **then** valorNat(d) ∈ claves(dn) **else** valorString(d) ∈ claves(ds) **fi**

4.2.3. Función de abstracción

Abs : $e : estr \to base$ {Rep(e)}

$(\forall\ e : estr)$ Abs(e) $=_{\text{obs}}$ $db : base \iff$

- $tablas(db) =_{\text{obs}} claves(e.tablasTree) \wedge_L$
- $(\forall t : tabla)\ t \in e.tablas \Rightarrow_L dameTabla(nombre(t), db) =_{\text{obs}} t \wedge$
- $(\forall nt, nt' : nombreTabla)\ nt \in tablas(db) \wedge nt' \in tablas(db) \Rightarrow$
  $(hayJoin?(nt, nt', db) \iff def?(nt, e.vistasJoin) \wedge_L def?(nt', obtener(nt, e.vistasJoin))) \wedge_L$
- $(\forall nt, nt' : nombreTabla) hayJoin?(nt, nt', db) \Rightarrow$
  $obtener(nt', obtener(nt, e.vistasJoin)).campo =_{\text{obs}} campoJoin(nt, nt', db)$

4.3. Algoritmos

iNuevaDB() → *res*: **base**

$res.tablas \leftarrow vacio()$	// $\mathcal{O}(1)$
$res.tablasTree \leftarrow nuevoDiccLog()$	// $\mathcal{O}(1)$
$res.vistasJoin \leftarrow nuevoTrie()$	// $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Justificación: 4 * $\mathcal{O}(1) = \mathcal{O}(1)$

iAgregarTabla(**in/out** db: $base$, **in** t: $tabla$)

var $it : itConj(tabla)$
$it \leftarrow agregarRapido(db.tablas, t)$ // $\mathcal{O}(1)$
$definir(db.tablasTree, nombre(t), it)$ // $\mathcal{O}(|nombre(t)|) = \mathcal{O}(1)$
if $cardinal(db.tablas) == 1$ **then** // $\mathcal{O}(1)$
  $db.tablaMaxima = it$ // $\mathcal{O}(1)$
end if

Complejidad: $\mathcal{O}(1)$
Justificación: 4 * $\mathcal{O}(1) = \mathcal{O}(1)$

---

iInsertarEntrada(**in/out** db: $base$, **in** nt: $nombreTabla$, **in** r: $registro$)

var $t : tabla$
var $it : itConj(tabla)$
$t \leftarrow tabla(db, nt)$ // $\mathcal{O}(1)$
$it \leftarrow tablas(db)$ // $\mathcal{O}(1)$
$agregarRegistro(t, r)$ // $\mathcal{O}(L + log(n))$ si hay un índice nat, $\mathcal{O}(L)$ sino
if $accesos(siguiente(db.tablaMaxima)) < accesos(t)$ **then** // $\mathcal{O}(1)$
  $db.tablaMaxima = obtener(db.tablasTree, nt)$ // $\mathcal{O}(1)$
end if

while $haySiguiente(it)$ **do** // $\mathcal{O}(1)$
  // El loop se repite T veces
  var $t' : tabla$
  $t' \leftarrow siguiente(it)$ // $\mathcal{O}(1)$
  if $nombre(t') != nombre(t)$ **then** // $\mathcal{O}(|nombre(t)|) = \mathcal{O}(1)$
    if $definido?(db.vistasJoin, nombre(t))$ **then** // $\mathcal{O}(|nombre(t)|) = \mathcal{O}(1)$
      var $joinsT : diccTrie(nombreTabla, vistaJoin)$
      $joinsT \leftarrow obtener(db.vistasJoin, nombre(t))$ // $\mathcal{O}(|nombre(t)|) = \mathcal{O}(1)$
      if $definido?(joinsT, nombre(t'))$ **then** // $\mathcal{O}(|nombre(t')|) = \mathcal{O}(1)$
        var $b : buffer$
        $b \leftarrow obtener(joinsT, nombre(t')).buffer$ // $\mathcal{O}(|nombre(t')|) = \mathcal{O}(1)$
        $agregarAtras(b, < esInsercion : true, tabla : false, reg : r >)$ // $\mathcal{O}(copy(registro)) = \mathcal{O}(L)$
      end if
    end if

    if $definido?(db.vistasJoin, nombre(t'))$ **then** // $\mathcal{O}(|nombre(t')|) = \mathcal{O}(1)$
      var $joinsT' : diccTrie(nombreTabla, vistaJoin)$
      $joinsT' \leftarrow obtener(db.vistasJoin, nombre(t'))$ // $\mathcal{O}(|nombre(t')|) = \mathcal{O}(1)$
      if $definido?(joinsT', nombre(t))$ **then** // $\mathcal{O}(|nombre(t)|) = \mathcal{O}(1)$
        var $b : buffer$
        $b \leftarrow obtener(joinsT', nombre(t)).buffer$ // $\mathcal{O}(|nombre(t)|) = \mathcal{O}(1)$
        $agregarAtras(b, < esInsercion : true, tabla : true, reg : r >)$ // $\mathcal{O}(copy(registro)) = \mathcal{O}(L)$
      end if
    end if
  end if
  $avanzar(it)$ // $\mathcal{O}(1)$
end while

Complejidad: $\mathcal{O}(T * L + log(n))$ si hay un índice nat, $\mathcal{O}(T * L)$ sino
Justificación: Si hay un índice nat: 2 * $\mathcal{O}(1)$ + $\mathcal{O}(L + log(n))$ + 2 * $\mathcal{O}(1)$ + T * (6 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + 4 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$) = $\mathcal{O}(T * L + log(n))$
Sino: 2 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + 2 * $\mathcal{O}(1)$ + T * (6 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + 4 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$) = $\mathcal{O}(T * L)$

iBorrar(**in/out** db: $base$, **in** nt: $nombreTabla$, **in** c: $campo$, **in** d: $dato$)

```
var t : tabla
var it : itConj(tabla)
var rs : conj(registro)
t ← tabla(db, nt)                                                          // O(1)
it ← tablas(db)                                                            // O(1)
rs ← borrarRegistro(t, c, d)          // O(L + log(n)) si c es índice, O(L * n) sino

if !esVacio(rs) then                                                       // O(1)
    var r : registro
    r ← siguiente(crearIt(rs))                                             // O(1)

    if accesos(siguiente(db.tablaMaxima)) < accesos(t) then                // O(1)
        db.tablaMaxima = obtener(db.tablasTree, nt)                        // O(1)
    end if

    while haySiguiente(it) do                                              // O(1)
        // El loop se repite T veces
        var t' : tabla
        t' ← siguiente(it)                                                 // O(1)
        if nombre(t') ! = nombre(t) then                                   // O(|nombre(t)|) = O(1)
            if definido?(db.vistasJoin, nombre(t)) then                    // O(|nombre(t)|) = O(1)
                var joinsT : diccTrie(nombreTabla, vistaJoin)
                joinsT ← obtener(db.vistasJoin, nombre(t))                 // O(|nombre(t)|) = O(1)
                if definido?(joinsT, nombre(t')) then                      // O(|nombre(t')|) = O(1)
                    var b : buffer
                    b ← obtener(joinsT, nombre(t')).buffer                 // O(|nombre(t')|) = O(1)
                    agregarAtras(b, < esInsercion : false, tabla : false, reg : r >)   // O(copy(registro)) = O(L)
                end if
            end if

            if definido?(db.vistasJoin, nombre(t')) then                   // O(|nombre(t')|) = O(1)
                var joinsT' : diccTrie(nombreTabla, vistaJoin)
                joinsT' ← obtener(db.vistasJoin, nombre(t'))               // O(|nombre(t')|) = O(1)
                if definido?(joinsT', nombre(t)) then                      // O(|nombre(t)|) = O(1)
                    var b : buffer
                    b ← obtener(joinsT', nombre(t)).buffer                 // O(|nombre(t)|) = O(1)
                    agregarAtras(b, < esInsercion : false, tabla : true, reg : r >)    // O(copy(registro)) = O(L)
                end if
            end if
        end if
        avanzar(it)                                                        // O(1)
    end while
end if
```

Complejidad: $\mathcal{O}(T * L + log(n))$ si c es un índice, $\mathcal{O}(L * (T + n))$ sino

Justificación: Si hay un índice nat: 2 * $\mathcal{O}(1)$ + $\mathcal{O}(L + log(n))$ + 4 * $\mathcal{O}(1)$ + T * (6 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + 4 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$) = $\mathcal{O}(T * L + log(n))$

Sino: 2 * $\mathcal{O}(1)$ + $\mathcal{O}(L*n)$ + 4 * $\mathcal{O}(1)$ + T * (6 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + 4 * $\mathcal{O}(1)$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$) = $\mathcal{O}(L*(T+n))$

iGenerarVistaJoin(**in/out** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$, **in** c: $campo$
) → res: **itConj(registro)**

var $recorreTabla1 : bool$
var $t_1, t_2 : tabla$
var $T_{recorro}, T_{busco} : tabla$
var $it : itConj(registro)$
var $v : vistaJoin$
$t_1 \leftarrow tabla(db, nt_1)$ // $\mathcal{O}(1)$
$t_2 \leftarrow tabla(db, nt_2)$ // $\mathcal{O}(1)$
$v \leftarrow < campo : c, tipo : tipoCampo(t_1, c)\ buffer : vacia(),$
$joins : vacio(), joinsNat : nuevoDiccLog(), joinString : nuevoTrie() >$ // $\mathcal{O}(1)$

// Generamos los joins recorriendo linealmente una tabla y buscando el índice correnspondiente en la otra.
// Para mantener el mejor orden de complejidad posible, si alguna tabla tiene un índice clave,
// la seleccionamos como "tabla a buscarτ recorremos linealmente la otra
// Si ambas tablas tienen índices clave, recorremos la que tenga la menor cantidad de elementos
// (Por el pre, c es siempre clave)
if $esIndice(t_1, c)$ && $esIndice(t_2, c)$ **then** // $\mathcal{O}(1)$
  if $cantidadRegistros(t_1) < cantidadRegistros(t_2)$ **then** // $\mathcal{O}(1)$
    $recorreTabla1 \leftarrow true$ // $\mathcal{O}(1)$
    $t_a \leftarrow t_1$ // $\mathcal{O}(1)$
    $t_b \leftarrow t_2$ // $\mathcal{O}(1)$
  else
    $recorreTabla1 \leftarrow false$ // $\mathcal{O}(1)$
    $t_a \leftarrow t_2$ // $\mathcal{O}(1)$
    $t_b \leftarrow t_1$ // $\mathcal{O}(1)$
  end if
else if $esIndice(t_1, c)$ **then** // $\mathcal{O}(1)$
  $recorreTabla1 \leftarrow true$ // $\mathcal{O}(1)$
  $t_a \leftarrow t_1$ // $\mathcal{O}(1)$
  $t_b \leftarrow t_2$ // $\mathcal{O}(1)$
else
  $recorreTabla1 \leftarrow false$ // $\mathcal{O}(1)$
  $t_a \leftarrow t_2$ // $\mathcal{O}(1)$
  $t_b \leftarrow t_1$ // $\mathcal{O}(1)$
end if
// t_a es t_1 si ambas tablas tienen índice clave en c y t_1 tiene la menor cantidad de elementos
// o solo t_2 tiene índice en c, t_2 en otro caso
// t_b es la otra tabla

$it \leftarrow registros(t_a)$ // $\mathcal{O}(1)$
while $haySiguiente(it)$ **do** // $\mathcal{O}(1)$
  // El loop se repite $|registros(t_a)|$ veces
  var $d : dato$
  var $rs : conj(registro)$
  var $r, r_a, r_b : registro$
  var $it_{reg} : itConj(registro)$
  $d \leftarrow significado(r_a, c)$ // $\mathcal{O}(1)$
  $rs \leftarrow buscar(t_b, definir(nuevoDiccLog(), c, d))$
  // $\mathcal{O}(L + log(|registros(t_b)|))$ si hay un índice clave sobre t_b, $\mathcal{O}(L * |registros(t_b)|)$ sino

(Continúa) iGenerarVistaJoin(**in/out** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$, **in** c: $campo$) → res: **itConj(registro)**

if !$esVacio?(rs)$ **then** // $\mathcal{O}(1)$
  $r_b \leftarrow siguiente(crearIt(rs))$ // $\mathcal{O}(1)$

  // Los valores de t_1 pisan a los de t_2 si hay campos repetidos
  if $recorreTabla1$ **then**
    $r \leftarrow combinarRegistros(r_a, r_b)$ // $\mathcal{O}(1)$
  else
    $r \leftarrow combinarRegistros(r_b, r_a)$ // $\mathcal{O}(1)$
  end if
  $it_{reg} \leftarrow agregarRapido(v.joins, r)$ // $\mathcal{O}(1)$

  if $v.tipo$ **then**
    $definir(v.joinsNat, valorNat(d), it_{reg})$ // $\mathcal{O}(log(n+m) * cmp(nat) + copy(it)) = \mathcal{O}(log(n+m))$
  else
    $definir(v.joinsString, valorString(d), it_{reg})$ // $\mathcal{O}(L + copy(it)) = \mathcal{O}(L)$
  end if
end if
$avanzar(it)$ // $\mathcal{O}(1)$
end while

if !$definido?(db.vistasJoin, nt_1)$ **then** // $\mathcal{O}(|nt_1|) = \mathcal{O}(1)$
  $definir(db.vistasJoin, nt_1, nuevoTrie())$ // $\mathcal{O}(|nt_1|) = \mathcal{O}(1)$
end if
$definir(obtener(db.vistasJoin, nt_1), nt_2, v)$ // $\mathcal{O}(|nt_2|) = \mathcal{O}(1)$
$res \leftarrow crearIt(v.joins)$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(min(n,m) * (L + log(n+m)))$ si el campo es indice de ambas tablas,
$\mathcal{O}(n * (L + log(n+m)))$ si el campo es solo índice de t_2,
$\mathcal{O}(m * (L + log(n+m)))$ si el campo es solo índice de t_1 y
$\mathcal{O}(m * (n * L + log(n)))$ si el campo no es un índice,
donde $n = |registros(t_1)|$ y $m = |registros(t_2)|$

Justificación: Si c es índice en ambas tablas: $|registros(t_a)|$ = min(n,m) y $|registros(t_b)|$ = max(n,m) ⇒
10 * $\mathcal{O}(1)$ + min(n,m) *
(2 * $\mathcal{O}(1)$ + $\mathcal{O}(L + log(max(n,m)))$ + 5 * $\mathcal{O}(1)$ + $\mathcal{O}(log(n+m))$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$)
= $\mathcal{O}(min(n,m) * (L + log(max(n,m)) + log(n+m))) = \mathcal{O}(min(n,m) * (L + log(n+m)))$

Si c es solo índice en t_2: $|registros(t_a)|$ = n y $|registros(t_b)|$ = m ⇒
10 * $\mathcal{O}(1)$ + n * (2 * $\mathcal{O}(1)$ + $\mathcal{O}(L + log(m))$ + 5 * $\mathcal{O}(1)$ + $\mathcal{O}(log(n+m))$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$)
= $\mathcal{O}(n * (L + log(m) + log(n+m))) = \mathcal{O}(n * (L + log(n+m)))$

Si c es solo índice en t_1: $|registros(t_a)|$ = m y $|registros(t_b)|$ = n ⇒
10 * $\mathcal{O}(1)$ + m * (2 * $\mathcal{O}(1)$ + $\mathcal{O}(L + log(n))$ + 5 * $\mathcal{O}(1)$ + $\mathcal{O}(log(n+m))$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$)
= $\mathcal{O}(m * (L + log(n) + log(n+m))) = \mathcal{O}(m * (L + log(n+m)))$

Si c no es índice: $|registros(t_a)|$ = m y $|registros(t_b)|$ = n ⇒
10 * $\mathcal{O}(1)$ + m * (2 * $\mathcal{O}(1)$ + $\mathcal{O}(L * n)$ + 5 * $\mathcal{O}(1)$ + $\mathcal{O}(log(n+m))$ + $\mathcal{O}(L)$ + $\mathcal{O}(1)$)
= $\mathcal{O}(m * (L + L * n + log(n+m))) = \mathcal{O}(m * (n * L + log(n+m)))$

---

iHayTabla(**in** db: $base$, **in** nt: $nombreTabla$) → res: **bool**

$res \leftarrow definido?(db.tablasTree, nt)$ // $\mathcal{O}(|nt|) = \mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iTabla(**in** db: $base$, **in** nt: $nombreTabla$) → res: **tabla**

$res \leftarrow obtener(db.tablasTree, nt)$ // $\mathcal{O}(|nt|) = \mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iTablas(**in** db: $base$) → res: **itConj(tabla)**

$res \leftarrow crearIt(db.tablas)$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iTablaMaxima(**in** db: $base$) → res: **tabla**

$res \leftarrow siguiente(db.tablaMaxima)$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iHayJoin(**in** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$) → res: **bool**

$res \leftarrow definido?(db.vistasJoin, nt_1)$ && $definido?(obtener(db.vistasJoin, nt_1), nt_2)$
// $\mathcal{O}(|nt_1|) + \mathcal{O}(|nt_2|) = \mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iCampoJoin(**in** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$) → res: **campo**

$res \leftarrow obtener(obtener(db.vistasJoin, nt_1), nt_2).campo$ // $\mathcal{O}(|nt_1|) + \mathcal{O}(|nt_2|) = \mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iBorrarJoin(**in** db: $base$, **in** nt_1: $nombreTabla$, **in** nt_2: $nombreTabla$)

$borrar(obtener(db.vistasJoin, nt_1), nt_2)$ // $\mathcal{O}(|nt_1|) + \mathcal{O}(|nt_2|) = \mathcal{O}(1)$
if $!hayS iguiente(crearIt(obtener(db.vistasJoin, nt_1)))$ **then** // $\mathcal{O}(|nt_1|) = \mathcal{O}(1)$
  $borrar(db.vistasJoin, nt_1)$ // $\mathcal{O}(|nt_1|) = \mathcal{O}(1)$
end if

Complejidad: $\mathcal{O}(1)$

Justificación: 3 * $\mathcal{O}(1) = \mathcal{O}(1)$

---

iVistaJoin(**in** db: $base$, **in** nt_1 : $nombreTabla$, **in** nt_2 : $nombreTabla$) $\rightarrow$ res: **itConj(registro)**

var $v : vistaJoin$
var $it : itLista(operacionJoin)$
var $t_1, t_2 : tabla$
$v \leftarrow obtener(obtener(db.vistasJoin, nt_1), nt_2)$ // $\mathcal{O}(|nt_1|) + \mathcal{O}(|nt_2|) = \mathcal{O}(1)$
$t_1 \leftarrow tabla(db, nt_1)$ // $\mathcal{O}(1)$
$t_2 \leftarrow tabla(db, nt_2)$ // $\mathcal{O}(1)$

$it \leftarrow crearIt(v.buffer)$ // $\mathcal{O}(1)$
while $haySiguiente(it)$ **do** // $\mathcal{O}(1)$
  // El loop se repite R veces
  var $op : operacionJoin$
  var $d : dato$
  $op \leftarrow siguiente(it)$ // $\mathcal{O}(1)$
  $d \leftarrow obtener(op.reg, v.campo)$ // $\mathcal{O}(1)$
  if $op.esInsercion$ **then**
    var $rs : conj(registro)$
    var $r, r_b : registro$
    // c siempre es clave (por el pre), por lo que buscar tendrá complejidad $\mathcal{O}(L + log(n))$ si c es índice
    if !$op.enTablaB$ **then** // $\mathcal{O}(1)$
      $rs \leftarrow buscar(t_2, definir(nuevoDiccLog(), v.campo, d))$ // $\mathcal{O}(L + log(m))$ si es índice, $\mathcal{O}(L * m)$ sino
    else
      $rs \leftarrow buscar(t_1, definir(nuevoDiccLog(), v.campo, d))$ // $\mathcal{O}(L + log(n))$ si es índice, $\mathcal{O}(L * n)$ sino
    end if
    if !$esVacio?(rs)$ **then** // $\mathcal{O}(1)$
      $r_b \leftarrow siguiente(crearIt(rs))$ // $\mathcal{O}(1)$

      if !$op.enTablaB$ **then** // $\mathcal{O}(1)$
        $r \leftarrow combinarRegistros(op.reg, r_b)$ // $\mathcal{O}(1)$
      else
        $r \leftarrow combinarRegistros(r_b, op.reg)$ // $\mathcal{O}(1)$
      end if

      $it_{reg} \leftarrow agregarRapido(v.joins, r)$ // $\mathcal{O}(copy(registro)) = \mathcal{O}(1)$
      if $v.tipo$ **then**
        $definirRapido(v.joinsNat, valorNat(d), it_{reg})$ // $\mathcal{O}(copy(it)) = \mathcal{O}(1)$
      else
        $definirRapido(v.joinsString, valorString(d), it_{reg})$ // $\mathcal{O}(copy(it)) = \mathcal{O}(1)$
      end if
    end if
  else
    if $v.tipo$ **then** // $\mathcal{O}(1)$
      if $definido?(v.joinsNat, valorNat(d))$ **then** // $\mathcal{O}(log(min(n,m)) * cmp(nat)) = \mathcal{O}(log(min(n,m)))$
        $eliminarSiguiente(obtener(v.joinsNat, valorNat(d)))$
        // $\mathcal{O}(log(min(n,m)) * cmp(nat)) = \mathcal{O}(log(min(n,m)))$
        $borrar(v.joinsNat, valorNat(d))$ // $\mathcal{O}(log(min(n,m)) * cmp(nat)) = \mathcal{O}(log(min(n,m)))$
      end if
    else
      if $definido?(v.joinsString, valorString(d))$ **then** // $\mathcal{O}(L)$
        $eliminarSiguiente(obtener(v.joinsString, valorString(d)))$ // $\mathcal{O}(L)$
        $borrar(v.joinsString, valorString(d))$ // $\mathcal{O}(L)$
      end if
    end if
  end if
  $avanzar(it)$ // $\mathcal{O}(1)$
end while
$res \leftarrow crearIt(v.joins)$ // $\mathcal{O}(1)$

(Continúa) iVistaJoin(**in** db: $base$, **in** nt_1 : $nombreTabla$, **in** nt_2 : $nombreTabla$) → res: **itConj(registro)**

Complejidad: $\mathcal{O}(1)$ si $R = 0$,
$\mathcal{O}(R * (L + log(n * m)))$ si $R > 0$ y ambas tablas tienen índice en el campo del join y
$\mathcal{O}(R * L * (n + m))$ sino
R es la cantidad de modificaciónes sobre las tablas desde la generación o última visualización del join

Justificación: Si $R = 0$: 6 * $\mathcal{O}(1) = \mathcal{O}(1)$

Si $R > 0$ y el campo es índice en ambas tablas: 4 * $\mathcal{O}(1)$ + R * (4 * $\mathcal{O}(1)$ + $\mathcal{O}(L + log(m))$
+ $\mathcal{O}(L + log(n))$ + 9 * $\mathcal{O}(1)$ + 3 * $\mathcal{O}(log(min(n, m)))$ + 3 * $\mathcal{O}(L)$ + $\mathcal{O}(1)$)
$= \mathcal{O}(R * (L + log(m) + log(n) + log(min(n, m)))) = \mathcal{O}(R * (L + log(m) + log(n)))$
$= \mathcal{O}(R * (L + log(m * n)))$

Si $R > 0$ y el campo no es índice en ambas tablas: 4 * $\mathcal{O}(1)$ + R * (4 * $\mathcal{O}(1)$ + $\mathcal{O}(L * m)$
+ $\mathcal{O}(L * n)$ + 9 * $\mathcal{O}(1)$ + 3 * $\mathcal{O}(log(min(n, m)))$ + 3 * $\mathcal{O}(L)$ + $\mathcal{O}(1)$)
$= \mathcal{O}(R * (L * m + L * n + log(min(n, m)) + L)) = \mathcal{O}(R * (L * (n + m) + log(n + m)))$
$= \mathcal{O}(R * L * (n + m))$

iBuscar(**in** db: $base$, **in** nt: $nombreTabla$, **in** r : $registro$) → res: **conj(registro)**

$res \leftarrow buscar(tabla(db, nt), r)$ // $\mathcal{O}(in)$

Complejidad: $\mathcal{O}(in)$, donde $in =$
$\mathcal{O}(L)$ si hay un campo clave e índice string como criterio,
$\mathcal{O}(L + log(n))$ si hay un campo clave e índice nat,
$\mathcal{O}(L * n)$ si no,
donde $n = |registros(dameTabla(nt, db)|)$

4.3.1. Funciones auxiliares de los algoritmos

CombinarRegistros(**in** r_1 : $registro$, **in** r_2 : $registro$) → res: **registro**
Pre ≡ {`true`}
Post ≡ {$res =_{\text{obs}} agregarCampos(r_1, r_2)$}
Complejidad: $\mathcal{O}(1)$
Descripción: La complejidad es constante pues la cantidad de campos está acotada

iCombinarRegistros(**in** r_1 : $registro$, **in** r_2 : $registro$) → res: **registro**

var it : $itDiccLog(campo, dato)$
$res \leftarrow vacio()$ // $\mathcal{O}(1)$

$it \leftarrow crearIt(r_2)$ // $\mathcal{O}(1)$
while $haySiguiente(it)$ **do** // $\mathcal{O}(1)$
// El loop se ejecuta $|campos(r_2)|$ veces, acotado por constante
$definir(res, siguiente(it).clave, siguiente(it).significado)$ // $\mathcal{O}(log(|campos(r_2)|)) = \mathcal{O}(1)$
end while

// Siguiendo la especificación, los valores de r_1 pisan los repetidos de r_2
$it \leftarrow crearIt(r_1)$ // $\mathcal{O}(1)$
while $haySiguiente(it)$ **do** // $\mathcal{O}(1)$
// El loop se ejecuta $|campos(r_1)|$ veces, acotado por constante
$definir(res, siguiente(it).clave, siguiente(it).significado)$ // $\mathcal{O}(log(|campos(r_2)| + |campos(r_1)|)) = \mathcal{O}(1)$
end while

Complejidad: $\mathcal{O}(1)$
Justificación: 2 * $\mathcal{O}(1)$ + k_2 * 2 * $\mathcal{O}(1)$ + $\mathcal{O}(1)$ + k_1 * 2 * $\mathcal{O}(1) = \mathcal{O}(1)$

4.4. Servicios usados

Se utilizan las operaciones exportadas por el módulo Dato con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo Tabla con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo DiccLog con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo DiccTrie con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo Conjunto Lineal (α) con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo Lista Enlazada (α) con sus complejidades declaradas.

5. diccTrie(α)

El módulo diccTrie provee un diccionario con claves de tipo String y acceso, inserción y borrado en $\mathcal{O}(L)$, donde L es el largo máximo de las claves.

5.1. Interfaz

parámetros formales

géneros α

No es necesario pedir operaciónes de comparación sobre los significados.

Operaciones

usa:

se explica con: Diccionario($string, \alpha$)

géneros: diccTrie(α)

operaciones:

NuevoDiccTrie() $\rightarrow$ *res*: **diccTrie(α)**
Pre $\equiv$ {`true`}
Post $\equiv \{res =_{\text{obs}} vacio\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un diccionario vacio

Definir(**in/out** d: `diccTrie(`α`)`, **in** c: `string`, **in** v: α)
Pre $\equiv \{d =_{\text{obs}} d_0 \wedge \neg def?(c, d)\}$
Post $\equiv \{d =_{\text{obs}} definir(c, v, d_0)\}$
Complejidad: $\mathcal{O}(L)$
Descripción: Modifica el diccionario agregando o reemplazando el significado de una clave con un nuevo valor

Definido?(**in** d: `diccTrie(`α`)`, **in** c: `string`) $\rightarrow$ *res*: **bool**
Pre $\equiv$ {`true`}
Post $\equiv \{res =_{\text{obs}} def?(c, d)\}$
Complejidad: $\mathcal{O}(L)$
Descripción: Devuelve true si una clave se encuentra definida en el diccionario

Obtener(**in** d: `diccTrie(`α`)`, **in** c: `string`) $\rightarrow$ *res*: α
Pre $\equiv \{def?(c, d)\}$
Post $\equiv \{res =_{\text{obs}} obtener(c, d)\}$
Complejidad: $\mathcal{O}(L)$
Descripción: Devuelve el significado definido para la clave c

Borrar(**in/out** d: `diccTrie(`α`)`, **in** c: `string`)
Pre $\equiv \{d =_{\text{obs}} d_0 \wedge def?(c, d)\}$
Post $\equiv \{d =_{\text{obs}} borrar(c, d_0)\}$
Complejidad: $\mathcal{O}(L)$
Descripción: Borra el significado asociado a la clave c

Maximo(**in** d: `diccLog(`α`)`) $\rightarrow$ *res*: **tupla(clave: string, significado: α)**
Pre $\equiv \{\neg(d =_{\text{obs}} vacio)\}$
Post $\equiv \{res =_{\text{obs}} tupla(maximo(d), obtener(maximo(d)))\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Obtiene una tupla de la clave y el significado del elemento con la clave mas grande en el diccionario

MINIMO(**in** d: `diccLog(`α`)`) $\rightarrow$ res: **tupla(clave: string, significado:** α**)**
Pre $\equiv \{\neg(d =_{\text{obs}} vacio)\}$
Post $\equiv \{res =_{\text{obs}} tupla(minimo(d), obtener(minimo(d)))\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Obtiene una tupla de la clave y el significado del elemento con la clave mas pequeña en el diccionario

5.2. Representación

`diccTrie(`α`)` **se representa con** `trie`

donde `trie` es `tupla(`
$raiz$: `puntero(nodo),`
$minimo$: `clavevalor,` $maximo$: `clavevalor`

donde `nodo` es `tupla(`
$valor$: α,
$hijos$: `arreglo_estático[256] (puntero(nodo)) )`

donde `clavevalor` es `tupla(`
$clave$: `string,`
$valor$: α `)`

5.2.1. Invariante de representación

Rep : estrDic $\rightarrow$ boolean

($\forall$ e: `estrDic`) Rep(e) $\equiv$ true $\iff$

- Existe un único camino entre cada nodo y el nodo raíz (no hay ciclos)
  $\neg(a.raiz = NULL) \Rightarrow noHayCiclos(ag(e.raiz, \emptyset), e.raiz)$
- Todos los nodos hojas, es decir, todos los que tienen su arreglo hijos con todas sus posiciones en NULL, tienen que tener un valor distinto de NULL.
  $\neg(a.raiz = NULL) \Rightarrow todasLasHojasTienenValor(e)$
- Los nodos minimo y maximo son los correspondientes
  $\neg(a.raiz = NULL) \Rightarrow (e.minimo == minimo(e.raiz) \wedge e.maximo == maximo(e.raiz))$

5.2.2. Operaciones auxiliares del invariante de Representación

noHayCiclos : conjunto(puntero(nodo)) p $\times$ puntero(nodo) q $\longrightarrow$ bool

noHayCiclos(p, q) $\equiv$ auxNoHayCiclos$(p, q, 0)$

auxNoHayCiclos : conjunto(puntero(nodo)) p $\times$ puntero(nodo) q $\times$ nat n $\longrightarrow$ bool

auxNoHayCiclos(p, q, n) $\equiv$ **if** $(*q).hijos[n] \in p$ **then**
false
else
if (*q).hijos[n]=NULL **then**
if $n < 255 \wedge$ auxNoHayCiclos$(p, q, n + 1)$ **then**
true
else
if n=255 **then** true **else** false **fi**
fi
else
if $n < 255 \wedge$ noHayCiclos$(ag(p, (*q).hijos[n]), (*q).hijos[n]) \wedge$
auxNoHayCiclos$(p, q, n + 1)$ **then**
true
else
if n=255 $\wedge$ noHayCiclos$(ag(p, (*q).hijos[n]), (*q).hijos[n])$ **then**
true
else
false
fi
fi
fi
fi

maximo : puntero(nodo) p $\longrightarrow$ clavevalor {noHayCiclos(ag($\emptyset$, p) , p)}

maximo(p) $\equiv$ **if** $\neg$tieneHijos(p) **then**
(NULL,p.valor)
else
(chr(auxMaximo(p,255))+maximo((*p).hijos[auxMaximo(p,255)]).clave),
(*p).hijos[auxMaximo(p,255)]).valor)
fi

auxMaximo : puntero(nodo) $p \times$ nat n $\longrightarrow$ nat {tieneHijos(p)}

auxMaximo(p, n) $\equiv$ **if** (*p).hijos[n] = NULL **then** auxMinimo(p,n-1) **else** n **fi**

minimo : puntero(nodo) p $\longrightarrow$ clavevalor {noHayCiclos(ag($\emptyset$, p) , p)}

minimo(p) $\equiv$ **if** $\neg$tieneHijos(p) **then**
(NULL,p.valor)
else
if (*p).valor = NULL **then**
(chr(auxMinimo(p,0))+minimo((*p).hijos[auxMinimo(p,0)]).clave),
(*p).hijos[auxMinimo(p,0)]).valor)
else
(NULL,p.valor)
fi
fi

auxMinimo : puntero(nodo) $p \times$ nat n $\longrightarrow$ nat {tieneHijos(p)}

auxMinimo(p, n) $\equiv$ **if** (*p).hijos[n] = NULL **then** auxMinimo(p,n+1) **else** n **fi**

todasLasHojasTienenValor : trie e $\longrightarrow$ bool {noHayCiclos(ag($\emptyset$, e.raiz) , e.raiz)}

todasLasHojasTienenValor(e) $\equiv$ **if** e.raiz = NULL **then** true **else** auxTodasLasHojasTienenValor$(e.raiz, 0)$ **fi**

auxTodasLasHojasTienenValor : puntero(nodo) $p \times$ nat n $\longrightarrow$ bool {noHayCiclos(ag($\emptyset$, p) , p)}

auxTodasLasHojasTienenValor(p, n) $\equiv$ **if** $\neg tieneHijos(p)$ **then**
$\neg((*p).valor = NULL)$
else
if (*p).hijos[n]=NULL **then**
$(n < 255 \wedge$ auxTodasLasHojasTienenValor$(p, n+1)) \vee$ n=255
else
if $n < 255 \wedge_L$ auxTodasLasHojasTienenValor$((*p).hijos[n], 0) \wedge$
auxTodasLasHojasTienenValor$(p, n+1)$ **then**
true
else
(n=255 $\wedge_L$ auxTodasLasHojasTienenValor$((*p).hijos[n], 0)$)
fi
fi
fi

tieneHijos : puntero(nodo) p $\longrightarrow$ bool

tieneHijos(p) $\equiv$ auxTieneHijos($p, 0$)

auxTieneHijos : puntero(nodo) $p \times$ nat n $\longrightarrow$ bool

auxTieneHijos(p, n) $\equiv$ **if** $n < 255 \wedge$ (*p).hijos[n]=NULL $\wedge \neg$auxTieneHijos($p, n+1$) **then**
false
else
$\neg$(n = 255 $\wedge$ (*p).hijos[n] = NULL)
fi

5.2.3. Función de abstracción

Abs : e : `estrDicc` $\rightarrow$ dicc(string,α) {Rep(e)}

($\forall$ e : `estrDicc`) Abs(e) $=_{\text{obs}}$ d : `dicc(string,`α`)` $\Longleftrightarrow$

- $(\forall s : string)\ def?(s, d) \Leftrightarrow (\neg(e.raiz = NULL) \wedge_L existe(s, 0, e.raiz)) \wedge_L$
- $(\forall s : string)\ def?(s, d) \Rightarrow obtener(s, d) = obtener(s, 0, e.raiz)$

existe : string $s \times$ nat $k \times$ puntero(nodo) n $\longrightarrow$ bool

existe(s, k, n) $\equiv$ **if** (*n).hijos[ord(s[k])] = NULL **then**
false
else
if k = long(s)-1 $\wedge \neg$((*n).hijos[ord(s[k])] = NULL) **then**
true
else
existe(s,k+1, (*n).hijos[ord(s[k])])
fi
fi

obtener : string $s \times$ nat $k \times$ puntero(nodo) n $\longrightarrow$ α {existe(s, 0, n)}

obtener(s, 0, n) $\equiv$ **if** k = long(s)-1 **then**
(*((*n).hijos[ord(s[k])])).valor
else
obtener(s,k+1, (*n).hijos[ord(s[k])])
fi

5.2.4. Representacion del iterador de Claves del diccTrie(α)

`itClaves(`α`)` **se representa con** `puntero(nodo)` Su Rep y Abs son los de itLista(α) definido en el apunte de iteradores para el modulo Lista Enlazada.

5.3. Algoritmos

iNuevoDiccTrie( ) → *res*: **diccTrie(α)**

$res.raiz \leftarrow NULL$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iDefinir(**in/out** e: `diccTrie(`α`)`, **in** c: `string`, **in** v: α)

var $i : nat$
$i \leftarrow 0$ // $\mathcal{O}(1)$
var $p : puntero(nodo)$
var $n : nodo$
var $eraVacio : bool$
if $e.raiz == NULL$ **then** // $\mathcal{O}(1)$
  $n \leftarrow crearNodo()$ // $\mathcal{O}(1)$
  $e.raiz \leftarrow \&n$ // $\mathcal{O}(1)$
  $eraVacio \leftarrow true$ // $\mathcal{O}(1)$
else
  $eraVacio \leftarrow false$ // $\mathcal{O}(1)$
end if
$p \leftarrow e.raiz$ // $\mathcal{O}(1)$
while $i < (longitud(c))$ **do**
  // El loop se repite longitud de la clave veces
  if $p.hijos[ord(s[i])] == NULL$ **then** // $\mathcal{O}(1)$
    $n \leftarrow crearNodo()$ // $\mathcal{O}(1)$
    $p.hijos[ord(s[i])] \leftarrow \&n$ // $\mathcal{O}(1)$
  end if
  $p \leftarrow p.hijos[ord(s[i])]$ // $\mathcal{O}(1)$
  $i++$ // $\mathcal{O}(1)$
end while
$*p.valor \leftarrow v$ // $\mathcal{O}(1)$
if $eraVacio \,||\, c < e.minimo.clave$ **then** // $\mathcal{O}(1)$
  $e.minimo.clave \leftarrow c$ // $\mathcal{O}(1)$
  $e.minimo.valor \leftarrow v$ // $\mathcal{O}(1)$
end if
if $eraVacio \,||\, c > e.maximo.clave$ **then** // $\mathcal{O}(1)$
  $e.maximo.clave \leftarrow c$ // $\mathcal{O}(1)$
  $e.maximo.valor \leftarrow v$ // $\mathcal{O}(1)$
end if
$AgregarAdelante(c, e.claves)$ // $\mathcal{O}(1)$
$p.clave \leftarrow crearIt(e.claves)$ // $\mathcal{O}(1)$

Justificación: 2* $\mathcal{O}(1)$ + L * (5 * $\mathcal{O}(1)$) + 7 * $\mathcal{O}(1)$ = $\mathcal{O}$(L)

iDefinido?(**in/out** e: `diccTrie(`α`)`, **in** c: `string`) $\rightarrow$ res: **bool**

```
var i : nat
var p : puntero(nodo)
i ← 0                                                                    // O(1)
if e.raiz == NULL then                                                   // O(1)
    res ← false                                                          // O(1)
else
    p ← e.raiz                                                           // O(1)
    while i < (longitud(c)) && ¬res do                                   // O(1)
                                  // El loop se repite longitud de la clave veces o menos en caso de que res se vuelva false
        if p.hijos[ord(s[i])]! = NULL then                               // O(1)
            p ← p.hijos[ord(s[i])]                                       // O(1)
        else
            res ← false                                                  // O(1)
        end if
        i + +                                                            // O(1)
    end while
    res ← p.valor! = NULL                                                // O(1)
end if
```

Complejidad: $\mathcal{O}$(L)

Justificación: 2* $\mathcal{O}(1)$ + L * (4 * $\mathcal{O}(1)$) + $\mathcal{O}(1)$ = $\mathcal{O}$(L)

iObtener(**in/out** e: `diccTrie(`α`)`, **in** c: `string`) $\rightarrow$ res: α

```
var i : nat
i ← 0                                                                    // O(1)
var p : puntero(nodo)
p ← e.raiz                                                               // O(1)
while i < (longitud(c)) do
                                                  // El loop se repite longitud de la clave veces
    p ← p.hijos[ord(s[i]))                                               // O(1)
    i + +                                                                // O(1)
end while
res ← (∗p).valor)                                                        // O(1)
```

Complejidad: $\mathcal{O}$(L)

Justificación: 2* $\mathcal{O}(1)$ + L * (2 * $\mathcal{O}(1)$) + $\mathcal{O}(1)$ = $\mathcal{O}$(L)

iBorrar(**in/out** e: `diccTrie(`α`)`, **in** c: `string`)

var i : nat
var p : $puntero(nodo)$
var pi : $pila(tupla(punte : puntero(nodo), siguiente : nat))$
$i \leftarrow 0$ // $\mathcal{O}(1)$
$p \leftarrow e.raiz$ // $\mathcal{O}(1)$
while $i < (longitud(c) - 1)$ **do** // $\mathcal{O}(1)$
// El loop se repite longitud de la clave veces
  $p \leftarrow p.hijos[ord(s[i]))$ // $\mathcal{O}(1)$
  $apilar(pi, (p, s[i + 1]))$ // $\mathcal{O}$(copy(tupla(punte: puntero(nodo), siguiente: nat)))
  $i++$ // $\mathcal{O}(1)$
end while
$EliminarSiguiente(p.clave)$ // $\mathcal{O}(1)$
$p.clave \leftarrow NULL$ // $\mathcal{O}(1)$
$p \leftarrow p.hijos[ord(s[i]))$ // $\mathcal{O}(1)$
$apilar(pi, (p, NULL))$ // $\mathcal{O}$(copy(tupla(punte: puntero(nodo), siguiente: nat)))
while $(\neg TieneHijos(*(tope(pi).punte))$ && $((*(tope(pi).punte)).clave == NULL)$ **do** // $\mathcal{O}(1)$
// El loop se repite longitud de la clave veces como mucho
  $p \leftarrow desapilar(pi)$ // $\mathcal{O}(1)$
  $(*(tope(pi).punte)).hijos[*(tope(pi).siguiente)] == NULL$ // $\mathcal{O}(1)$
end while
if $\neg tieneHijos(e.raiz)$ **then** // $\mathcal{O}(1)$
  $e.raiz \leftarrow NULL$ // $\mathcal{O}(1)$
else
  $p \leftarrow e.raiz$ // $\mathcal{O}(1)$
  var s : $string$
  $s \leftarrow <>$ // $\mathcal{O}(1)$
  while $(*p).valor == NULL$ **do** // $\mathcal{O}(1)$
// El loop se repite L veces como mucho
    $s \leftarrow s + chr(MenorHijo(*p))$ // $\mathcal{O}(1)$
    $p \leftarrow MenorHijo(*p)$ // $\mathcal{O}(1)$
  end while
  $e.minimo.clave \leftarrow s$ // $\mathcal{O}(1)$
  $e.minimo.valor \leftarrow (*p).valor$ // $\mathcal{O}(1)$
  $s \leftarrow <>$ // $\mathcal{O}(1)$
  while $TieneHijos(*p)$ **do** // $\mathcal{O}(1)$
// El loop se repite L veces como mucho
    $s \leftarrow s + chr(MayorHijo(*p))$ // $\mathcal{O}(1)$
    $p \leftarrow MayorHijo(*p)$ // $\mathcal{O}(1)$
  end while
  $e.maximo.clave \leftarrow s$ // $\mathcal{O}(1)$
  $e.maximo.valor \leftarrow (*p).valor$ // $\mathcal{O}(1)$
end if

Complejidad: $\mathcal{O}$(L)

Justificación: 2* $\mathcal{O}(1)$ + L * (2 * $\mathcal{O}(1)$ + $\mathcal{O}$(copy(tupla(punte: puntero(nodo), siguiente: nat)))) + $\mathcal{O}(1)$ +$\mathcal{O}$(copy(tupla(punte: puntero(nodo), siguiente: nat))))+ L * 2 * $\mathcal{O}(1)$+ 7 * $\mathcal{O}(1)$== $\mathcal{O}$(L) en caso de que el arbol este vacio, el el otro caso es 2* $\mathcal{O}(1)$ + L * (2 * $\mathcal{O}(1)$ + $\mathcal{O}$(copy(tupla(punte: puntero(nodo), siguiente: nat)))) + $\mathcal{O}(1)$ +$\mathcal{O}$(copy(tupla(punte: puntero(nodo), siguiente: nat))))+ L * 2 * $\mathcal{O}(1)$+$\mathcal{O}(1)$+2* L * 2 * $\mathcal{O}(1)$ +2 * 2 *big O(1) == $\mathcal{O}$(L)

---

iMaximo(**in/out** e: `diccTrie(`α`)`) $\rightarrow$ res: **tupla(clave: string, significado:** α**)**

$res \leftarrow e.maximo$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iMinimo(**in/out** e : `diccTrie(`α`)`) $\rightarrow$ *res*: **tupla(clave: string, significado:** α**)**

res $\leftarrow$ *e.minimo* // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

5.3.1. Funciones auxiliares de los algoritmos

iCrearNodo( ) $\rightarrow$ *res*: **nodo**

var d : $arreglo_est\acute{a}tico[256](puntero(nodo))$
var i : *nat*
$i \leftarrow 0$) // $\mathcal{O}(1)$
while $i < 256$ **do**
// El loop se repite 256 veces
  $d[i] \leftarrow NULL$ // $\mathcal{O}(1)$
  $i++$ // $\mathcal{O}(1)$
end while
res.hijos $\leftarrow d$ // $\mathcal{O}(1)$
res.valor $\leftarrow NULL$ // $\mathcal{O}(1)$
res.clave $\leftarrow NULL$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: $\mathcal{O}(1)$ + 256 * (2 * $\mathcal{O}(1)$) + 3 * $\mathcal{O}(1)$ = $\mathcal{O}(1)$

iTieneHijos(**in** n : `nodo`) $\rightarrow$ *res*: **bool**

var i : *nat*
$i \leftarrow 0$ // $\mathcal{O}(1)$
res $\leftarrow false$ // $\mathcal{O}(1)$
while $i < 256$ **do**
// El loop se repite 256 veces
  if $n.hijos[i] != NULL$ **then** // $\mathcal{O}(1)$
    res $\leftarrow true$ // $\mathcal{O}(1)$
  end if
end while

Complejidad: $\mathcal{O}(1)$
Justificación: 2* $\mathcal{O}(1)$ + 256 * (2 * $\mathcal{O}(1)$) = $\mathcal{O}(1)$

iMenorHijo(**in** n : `nodo`) $\rightarrow$ *res*: **nat**

var i : *nat*
$i \leftarrow 0$ // $\mathcal{O}(1)$
res $\leftarrow 256$ // $\mathcal{O}(1)$
while $i < 256$ **do**
// El loop se repite 256 veces
  if $n.hijos[i] != NULL$ && $i < res$ **then** // $\mathcal{O}(1)$
    res $\leftarrow i$ // $\mathcal{O}(1)$
  end if
end while

Complejidad: $\mathcal{O}(1)$
Justificación: 2* $\mathcal{O}(1)$ + 256 * (2 * $\mathcal{O}(1)$) = $\mathcal{O}(1)$

iMayorHijo(**in** n : `nodo`) $\rightarrow$ res: **nat**

```
var i : nat
i ← 0                                                          // O(1)
res ← 0                                                        // O(1)
while i < 256 do
                                                               // El loop se repite 256 veces
    if n.hijos[i]! = NULL && i > res then                      // O(1)
        res ← i                                                // O(1)
    end if
end while
```

Complejidad: $\mathcal{O}(1)$

Justificación: 2* $\mathcal{O}(1)$ + 256 * (2 * $\mathcal{O}(1)$) = $\mathcal{O}(1)$

5.4. Servicios usados

Se utilizan las operaciones exportadas por el módulo pila (α) con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo arreglo_estatico (α) con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo AgregarAdelante(itLista(α)) con sus complejidades declaradas.

Se utilizan las operaciones exportadas por el módulo EliminarSiguiente(itLista(α)) con sus complejidades declaradas.

6. diccLog(κ, α)

El módulo diccLog provee un diccionario con acceso, inserción y borrado en $\mathcal{O}(\log(n))$, donde n es la cantidad de elementos actuales.

6.1. Interfaz

parámetros formales

géneros κ, α

función $\bullet = \bullet$ (**in** $k_0 : \kappa$, **in** $k_1 : \kappa$) $\rightarrow$ *res*: **bool**
Pre $\equiv$ {`true`}
Post $\equiv$ {`res` $=_{\text{obs}}$ $(k_0 = k_1)$}
Complejidad: $\Theta(equal(k_0, k_1))$
Descripción: Función de igualdad de κ's

función $\bullet > \bullet$ (**in** $k_0 : \kappa$, **in** $k_1 : \kappa$) $\rightarrow$ *res*: **bool**
Pre $\equiv$ {`true`}
Post $\equiv$ {`res` $=_{\text{obs}}$ $(k_0 > k_1)$}
Complejidad: $\Theta(greater(k_0, k_1))$
Descripción: Función orden estricto de κ's

función COPIAR (**in** $k : \kappa$) $\rightarrow$ *res*: κ
Pre $\equiv$ {`true`}
Post $\equiv$ {$res =_{\text{obs}} k$}
Complejidad: $\mathcal{O}(copy(k))$
Descripción: Funcion de copia de κ's

Operaciones

usa: Lista Enlazada(α)

se explica con: Diccionario(κ, α), Iterador Unidireccional Modificable(tupla(κ, α))

géneros: diccLog(κ, α), itDiccLog(κ, α)

operaciones:

NUEVODICCLOG() $\rightarrow$ *res*: **diccLog(κ, α)**
Pre $\equiv$ {`true`}
Post $\equiv$ {$res =_{\text{obs}} vacio$}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un diccionario vacio

DEFINIR(**in/out** d: `diccLog(`κ, α`)`, **in** $c : \kappa$, **in** $v : \alpha$)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}} definir(c, v, d_0)$}
Complejidad: $\mathcal{O}(log(n)cmp + copy(c))$, donde $n = \#(claves(d))$ y $cmp = equal(c, c') + greater(c, c')$
Aliasing: alias($obtener(c, d) = v$), hasta que se redefina o se borre la clave
Descripción: Modifica el diccionario agregando o reemplazando el significado de una clave con un nuevo valor

DEFINIDO?(**in** d: `diccLog(`κ, α`)`, **in** $c : \kappa$) $\rightarrow$ *res*: **bool**
Pre $\equiv$ {`true`}
Post $\equiv$ {$res =_{\text{obs}} def?(c, d)$}
Complejidad: $\mathcal{O}(log(n)cmp)$, donde $n = \#(claves(d))$ y $cmp = equal(c, c') + greater(c, c')$
Descripción: Devuelve true si una clave se encuentra definida en el diccionario

OBTENER(**in** d: `diccLog(`κ, α`)`, **in** c: κ) $\rightarrow$ *res*: α
Pre $\equiv \{def?(c, d)\}$
Post $\equiv \{res =_{\text{obs}} obtener(c, d)\}$
Complejidad: $\mathcal{O}(log(n)cmp)$, donde $n = \#(claves(d))$ y $cmp = equal(c, c') + greater(c, c')$
Descripción: Devuelve el significado definido para la clave c

BORRAR(**in/out** d: `diccLog(`κ, α`)`, **in** c: κ)
Pre $\equiv \{d =_{\text{obs}} d_0 \wedge def?(c, d)\}$
Post $\equiv \{d =_{\text{obs}} borrar(c, d_0)\}$
Complejidad: $\mathcal{O}(log(n)cmp)$, donde $n = \#(claves(d))$ y $cmp = equal(c, c') + greater(c, c')$
Descripción: Borra el significado asociado a la clave c

MAXIMO(**in** d: `diccLog(`κ, α`)`) $\rightarrow$ *res*: **tupla(clave:** κ**, significado:** α**)**
Pre $\equiv \{\neg(d =_{\text{obs}} vacio)\}$
Post $\equiv \{res =_{\text{obs}} tupla(maximo(d), obtener(maximo(d)))\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Obtiene una tupla de la clave y el significado del elemento con la clave mas grande en el diccionario

MINIMO(**in** d: `diccLog(`κ, α`)`) $\rightarrow$ *res*: **tupla(clave:** κ**, significado:** α**)**
Pre $\equiv \{\neg(d =_{\text{obs}} vacio)\}$
Post $\equiv \{res =_{\text{obs}} tupla(minimo(d), obtener(minimo(d)))\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Obtiene una tupla de la clave y el significado del elemento con la clave mas pequeña en el diccionario

minimo : diccLog(κ α) d $\longrightarrow$ κ $\quad \{\neg vacia?(claves(d))\}$

minimo(d) $\equiv$ **if** $len(claves(d)) = 1$ **then** $prim(claves(d))$ **else**
if $prim(claves(d)) > minimo(borrar(prim(claves(d)), d))$ **then**
$minimo(borrar(prim(claves(d)), d))$
else
$prim(claves(d))$
fi
fi

maximo : diccLog(κ α) d $\longrightarrow$ κ $\quad \{\neg vacia?(claves(d))\}$

maximo(d) $\equiv$ **if** $len(claves(d)) = 1$ **then** $prim(claves(d))$ **else**
if $prim(claves(d)) > maximo(borrar(prim(claves(d)), d))$ **then**
$prim(claves(d))$
else
$maximo(borrar(prim(claves(d)), d))$
fi
fi

6.1.1. Operaciones del iterador

CREARIT(**in** d: `diccLog(`κ, α`)`) $\rightarrow$ *res*: **itDiccLog(**κ, α**)**
Pre $\equiv \{$`true`$\}$
Post $\equiv \{res =_{\text{obs}} crearItMod(<>, listaDeTupas(d)) \wedge$ `alias(`$esPermutacion(SecuSuby(res), listaDeTuplas(d))$`)`$\}$
Complejidad: $\mathcal{O}(copy(\alpha))$
Aliasing: El iterador se invalida luego de cualquier operacion que modifique el arbol. No se puede asegurar con un dfs que el orden de los elementos en el arbol sea el mismo luego de un balanceo.

HAYMAS?(**in** it: `itDiccLog(`κ, α`)`) $\rightarrow$ *res*: **bool**
Pre $\equiv \{$`true`$\}$
Post $\equiv \{res =_{\text{obs}} hayMas?(it)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve true si hay elementos para avanzar

ACTUAL(**in** it : `itDiccLog(`κ, α`)`) $\rightarrow$ res: **tupla(clave:** κ**, significado:** α**)**
Pre $\equiv \{hayMas?(it)\}$
Post $\equiv \{res =_{\text{obs}} actual(it) \wedge$ `alias(`$res.significado =_{\text{obs}} d.obtener(res.clave)$`)`$\}$
Complejidad: $\mathcal{O}(1)$
Aliasing: res.significado es una referencia al significado en el diccionario d sobre el que se itera.

AVANZAR(**in/out** it : `itDiccLog(`κ, α`)`)
Pre $\equiv \{it =_{\text{obs}} it_0 \wedge hayMas?(it)\}$
Post $\equiv \{it =_{\text{obs}} avanzar(it_0)\}$
Complejidad: $\mathcal{O}(copy(\alpha))$
Descripción: Avanza a la siguiente posición del iterador

listaDeTuplas : diccLog(κ α) d $\longrightarrow$ secuencia(tupla(κ α))

listaDeTuplas(d) $\equiv$ **if** $len(claves(d)) = 0$ **then** $<>$ **else**
$< prim(claves(d)), obtener(prim(claves(d))) > \bullet listaDeTuplas(borrar(prim(claves(d)), d))$
fi

6.2. Representación del diccionario

`diccLog(`κ, α`)` **se representa con** `avl`

donde `avl` es `tupla(`
$raiz$`: puntero(nodo),`
max`: puntero(nodo),` min`: puntero(nodo) )`

donde `nodo` es `tupla(`
$clave$`:` κ`,` $valor$`:` α`,`
$menor$`: nodo,` $mayor$`: nodo,` $padre$`: nodo`
fdb`: nat )`

6.2.1. Invariante de representación

Rep : $avl \rightarrow$ boolean

$(\forall\ a : avl)$ Rep(a) $\equiv$ true $\iff$

- raiz es null (y max y min tambien son null) $\vee_L$
  $a.raiz = NULL \Rightarrow (a.max = NULL \wedge a.min = NULL)$
- para cada nodo el factor de balanceo esta entre -1 y 1
  $fdbsEnRango(a.raiz)$
- para cada nodo el factor de balanceo es igual a la diferencia de altura de sus dos hijos
  $fdbsCoinciden(a.raiz)$
- para cada nodo, si tiene un nodo menor, su valor es mayor al de todos los nodos (del maximo) de la rama menor
  $menoresSonMenores(a.raiz)$
- para cada nodo, si tiene un nodo mayor, su valor es menor al de todos los nodos (del minimo) de la rama mayor
  $mayoresSonMayores(a.raiz)$
- el padre de raiz siembre debe ser null
  $a.raiz \neq NULL \Rightarrow (*a.raiz).padre = NULL$
- para cada nodo n, si tiene mayor o menor, sus padres debe ser n
  $a.raiz \neq NULL \Rightarrow padresCorrectos(a.raiz)$
- para cada dos nodos distintos, n_1 y n_2, sus hijos deben ser distintos
  $verificarHijos(a.raiz, a.raiz)$
- para cada nodo ninguno puede apuntar a la raiz

- max es igual al nodo con la clave mas grande
  $a.raiz \neq NULL \ \Rightarrow \ a.max = \&maximo(a.raiz)$
- min es igual al nodo con la clave mas pequeña
  $a.raiz \neq NULL \ \Rightarrow \ a.min = \&minimo(a.raiz)$

fdbsEnRango : puntero(nodo) n $\longrightarrow$ bool

fdbsEnRango(n) $\equiv$ **if** $n = NULL$ **then** $true$ **else**
$-1 \leq (*n).fdb \leq 1 \wedge fdbsEnRango((*n).menor) \wedge fdbsEnRango((*n).mayor)$
fi

fdbsCoinciden : puntero(nodo) n $\longrightarrow$ bool

fdbsCoinciden(n) $\equiv$ **if** $n = NULL$ **then** $true$ **else**
$(*n).fdb = altura((*n).menor) - altura((*n).mayor)$
$\wedge \, fdbsCoinciden((*n).menor) \wedge fdbsCoinciden((*n).mayor)$
fi

atura : puntero(nodo) n $\longrightarrow$ nat

altura(n) $\equiv$ **if** $n = NULL$ **then** 0 **else**
$1 + max(altura((*n).menor), altura((*n).mayor))$
fi

menoresSonMenores : puntero(nodo) n $\longrightarrow$ bool

menoresSonMenores(n) $\equiv$ **if** $n = NULL \vee (*n).menor = NULL$ **then** $true$ **else**
$maximo((*n).menor).clave < (*n).clave$
$\wedge \, menoresSonMenores((*n).menor) \wedge menoresSonMenores((*n).mayor)$
fi

minimo : puntero(nodo) n $\longrightarrow$ nodo $\{n \neq NULL\}$

minimo(n) $\equiv$ **if** $(*n).menor = NULL$ **then** $(*n)$ **else**
$minimo((*n).menor)$
fi

mayoresSonMayores : puntero(nodo) n $\longrightarrow$ bool

mayoresSonMayores(n) $\equiv$ **if** $n = NULL \vee (*n).mayor = NULL$ **then** $true$ **else**
$minimo((*n).mayor).clave < (*n).clave$
$\wedge \, mayoresSonMayores((*n).menor) \wedge mayoresSonMayores((*n).mayor)$
fi

maximo : puntero(nodo) n $\longrightarrow$ nodo $\{n \neq NULL\}$

maximo(n) $\equiv$ **if** $(*n).mayor = NULL$ **then** $(*n)$ **else**
$maximo((*n).mayor)$
fi

padresCorrectos : puntero(nodo) n $\longrightarrow$ bool $\{n \neq NULL\}$

padresCorrectos(n) $\equiv$ $((*n).menor \neq NULL \ \Rightarrow \ ((*n).menor.padre = n \wedge padresCorrectos((*n).menor))$
$\wedge \, ((*n).mayor \neq NULL \ \Rightarrow \ ((*n).mayor.padre = n \wedge padresCorrectos((*n).mayor))$

verificarHijos : puntero(nodo) $r \times$ puntero(nodo) n $\longrightarrow$ bool

verificarHijos(r, n) $\equiv$ **if** $r = NULL \vee n = NULL$ **then** $true$ **else**
$(r \neq n \ \Rightarrow \ hijosDistintos(*r, *n))$
$\wedge \, verificarHijos((*r).menor, n)$
$\wedge \, verificarHijos((*r).mayor, n)$
$\wedge \, verificarHijos(r, (*n).menor)$
$\wedge \, verificarHijos(r, (*n).mayor)$
fi

hijosDistintos : nodo $a \times$ nodo b $\longrightarrow$ bool

hijosDistintos(a, b) $\equiv$ $a.menor \neq NULL \Rightarrow$
$(a.menor \neq a.mayor \wedge a.menor \neq b.menor \wedge a.menor \neq b.mayor)$
$a.mayor \neq NULL \Rightarrow$
$(a.mayor \neq b.menor \wedge a.mayor \neq b.mayor)$
$b.menor \neq NULL \Rightarrow$
$(b.menor \neq b.mayor)$

6.2.2. Función de abstracción

Abs : $a : avl \rightarrow dicc$ {Rep(a)}

$(\forall\, a : avl)$ Abs(a) $=_{\text{obs}}$ $d : dicc \iff$

- $(\forall c : \kappa)\ def?(c, d) \Leftrightarrow existe(c, a.raiz) \wedge_L$
- $(\forall c : \kappa)\ def?(c, d) \Rightarrow obtener(c, d) = obtener(c, a.raiz)$

existe : $\kappa\ c \times$ puntero(nodo) $n \longrightarrow$ bool

existe(c, n) $\equiv$ **if** $n = NULL$ **then** $false$ **else**
$(*n).clave = c \vee_L existe(c, (*n).menor) \vee existe(c, (*n).mayor)$
fi

obtener : $\kappa\ c \times$ puntero(nodo) $n \longrightarrow \alpha$ {existe(c, n)}

obtener(c, n) $\equiv$ **if** $(*n).clave = c$ **then** $(*n).valor$ **else**
if $existe(c, (*n).menor)$ **then**
$obtener(c, (*n).menor)$
else
$obtener(c, (*n).mayor)$
fi
fi

6.3. Representación del iterador

`itDiccLog(`κ, α`)` **se representa con** `iter`

donde `iter` es `tupla(`
actual`: puntero(nodo),`
siguientes`: lista(puntero(nodo)),`
dicc`: puntero(avl)`

6.3.1. Invariante de representación del iterador

Rep : $iter \rightarrow$ boolean

$(\forall\, it : iter)$ Rep(it) $\equiv$ true $\iff$

- El puntero al diccionario no puede ser null
$it.dicc \neq NULL \wedge_L$
- Si el diccionario esta vacio no hay un elemento actual
$(*it.dicc).raiz = NULL \Rightarrow it.actual = NULL \wedge_L$
- Si actual es null entonces no puede haber siguientes
$it.actual = NULL \Rightarrow it.siguientes =<> \wedge_L$
- Si el diccionario no esta vacio y hay un elemento actual el elemento actual debe pertenecer al diccionario
$((*it.dicc).raiz \neq NULL \wedge it.actual \neq NULL) \Rightarrow$
$pertenece(it.actual, (*it.dicc).raiz)$
- Todos los elementos siguientes pertenecen al diccionario
$(\forall m : puntero(nodo), m \in it.siguientes) pertenece(m, (*it.dicc).raiz)$

- Si hay un actual, sus hijos pertenecen a los elementos siguientes
  $it.actual \neq NULL \Rightarrow$
  $(*it.actual).menor \neq NULL \Rightarrow (*it.actual).menor \in it.siguientes$
  $\wedge\, (*it.actual).menor \neq NULL \Rightarrow (*it.actual).menor \in it.siguientes$

pertenece : puntero(nodo) a × puntero(nodo) b $\longrightarrow$ bool $\{a \neq NULL\}$

pertenece(a, b) $\equiv$ **if** $b = NULL$ **then** $false$ **else**
$b = a \vee_L pertenece(a, (*b).menor) \vee pertenece(a, (*b).mayor$
fi

6.3.2. Función de abstracción del iterador

Abs : $it : iter \rightarrow itMod$ $\{\text{Rep}(it)\}$

$(\forall\, it : iter)$ Abs$(it) =_{\text{obs}} im : itMod \iff$

- $anteriores(im) =_{\text{obs}} hasta(it.actual, dfs((*it.dicc).raiz))$
- $siguientes(im) =_{\text{obs}} desde(it.actual, dfs((*it.dicc).raiz))$

hasta : $\gamma\ x$ × secuencia(γ) xs $\longrightarrow$ secuencia(γ)

hasta(x, xs) $\equiv$ **if** $vacia?(xs)$ **then** $<>$ **else**
if $prim(xs) = x$ **then** $<>$ **else**
$prim(xs) \bullet hasta(x, fin(xs))$
fi
fi

desde : $\gamma\ x$ × secuencia(γ) xs $\longrightarrow$ secuencia(γ)

desde(x, xs) $\equiv$ **if** $vacia?(xs)$ **then** $<>$ **else**
if $prim(xs) = x$ **then** xs **else**
$desde(x, fin(xs))$
fi
fi

dfs : puntero(nodo) n $\longrightarrow$ secuencia(puntero(nodo))

dfs(n) $\equiv$ **if** $n = NULL$ **then** $<>$ **else**
$n \bullet dfs((*n).menor)$ & $dfs((*n).mayor)$
fi

6.4. Algoritmos

iNuevoDiccLog() → *res*: **diccLog**	
$res.raiz \leftarrow NULL$	// $\mathcal{O}(1)$
$res.min \leftarrow NULL$	// $\mathcal{O}(1)$
$res.max \leftarrow NULL$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	
Justificación: 3 * $\mathcal{O}(1) = \mathcal{O}(1)$	

iMaximo(**in** d: `diccLog(`κ, α`)`) → *res*: **tupla(clave: κ, significado: α)**	
$res \leftarrow <clave : d.max.clave,\ significado : d.max.valor>$	// $\mathcal{O}(1)$
Complejidad: $\mathcal{O}(1)$	
Justificación: El acceso a las variables y la creacion de una tupla es $\mathcal{O}(1)$	

iMinimo(**in** d: `diccLog(`κ, α`)`) $\rightarrow$ res: **tupla(clave:** κ**, significado:** α**)**

$res \leftarrow < clave : d.min.clave,\ significado : d.min.valor >$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Justificación: El acceso a las variables y la creacion de una tupla es $\mathcal{O}(1)$

---

iDefinido?(**in** d: `diccLog(`κ, α`)`, **in** c: κ) $\rightarrow$ res: **bool**

$res \leftarrow iauxDefinidoNodo(d.raiz, c)$ // $\mathcal{O}(log(N)(greater(\kappa,\kappa) + equal(\kappa,\kappa)))$

Complejidad: $\mathcal{O}(log(N)(greater(\kappa,\kappa) + equal(\kappa,\kappa)))$

Justificación: Ver $iauxDefinidoNodo$.

---

iauxDefinidoNodo(**in** n: `puntero(nodo)`, **in** c: κ) $\rightarrow$ res: **bool**

if $n = NULL$ **then** // $\mathcal{O}(1)$
  $res \leftarrow false$ // $\mathcal{O}(1)$
else // $\mathcal{O}(1)$
  if $c = (*n).clave$ **then** // $\mathcal{O}(equal(\kappa,\kappa))$
    $res \leftarrow true$ // $\mathcal{O}(1)$
  else
    if $c > (*n).clave$ **then** // $\mathcal{O}(greater(\kappa,\kappa))$
      $res \leftarrow auxDefinidoNodo((*n).mayor, c)$ // $\mathcal{O}((log(N)-1)(greater(\kappa,\kappa) + equal(\kappa,\kappa)))$
    else
      $res \leftarrow auxDefinidoNodo((*n).menor, c)$ // $\mathcal{O}((log(N)-1)(greater(\kappa,\kappa) + equal(\kappa,\kappa)))$
    end if
  end if
end if

Complejidad: $\mathcal{O}(log(N)(greater(\kappa,\kappa) + equal(\kappa,\kappa)))$

Justificación: Donde N es la cantidad de elementos que hay insertados debajo de este nodo. En cada llamada a $auxDefinidoNodo$ se reduce la cantidad de elementos aproximadamente a la mitad haciendo una busqueda binaria y se compara la clave por igualdad y por orden. Luego de log(N) llamadas se llega a las hojas del arbol terminando la busqueda.

---

iObtener(**in** d: `diccLog(`κ, α`)`, **in** c: κ) $\rightarrow$ res: α

$res \leftarrow (*auxObtenerNodo(d.raiz, c)).valor$ // $\mathcal{O}(log(N)(greater(\kappa,\kappa) + equal(\kappa,\kappa)))$

Complejidad: $\mathcal{O}(log(N)(greater(\kappa,\kappa) + equal(\kappa,\kappa)))$

Justificación: Por complejidad ver $iauxObtenerNodo$. Como es condición necesaria para Obtener que la clave este definida se puede asumir que $auxObtenerNodo$ devuelve un puntero no nulo.

---

iauxObtenerNodo(**in** n : `puntero(nodo)`, **in** c : κ) $\rightarrow$ *res*: α

```
if n = NULL then                                                          // O(1)
    res ← NULL                                                            // O(1)
else                                                                      // O(1)
    if c = (*n).clave then                                                // O(equal(κ, κ))
        res ← n                                                           // O(1)
    else
        if c > (*n).clave then                                            // O(greater(κ, κ))
            res ← auxObtenerNodo((*n).mayor, c)                           // O((log(N) − 1)(greater(κ, κ) + equal(κ, κ)))
        else
            res ← auxObtenerNodo((*n).menor, c)                           // O((log(N) − 1)(greater(κ, κ) + equal(κ, κ)))
        end if
    end if
end if
```

Complejidad: $\mathcal{O}(log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$

Justificación: Al igual que con $iauxDefinidoNodo$, N es la cantidad de elementos debajo de este nodo. En cada llamada a $auxObtenerNodo$ se reduce la cantidad de elementos aproximadamente a la mitad y se compara la clave por igualdad y por orden.

---

iDefinir(**in/out** d : `diccLog(`κ, α`)`, **in** c : κ, **in** v : α)

```
if d.raiz = NULL then                                                     // O(1)
    d.raiz ← auxNuevoNodo(c, v)                                           // O(copy(κ))
else
    auxDefinirNodo(d.raiz, c, v)                                          // O(copy(κ) + log(N)(greater(κ, κ) + equal(κ, κ)))
end if
```

Complejidad: $\mathcal{O}(copy(\kappa) + log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$

Justificación: Donde N es la cantidad de elementos del arbol. La complejidad de definir una clave nueva es la de buscar si existe la misma ($log(N)$ comparaciones) y luego copiar la clave. El peor caso es en el que la clave no existe y hay que bajar hasta las hojas del arbol para insertar un nuevo nodo. Incluso si el arbol esta vacio entonces $log(N) = 0$ por lo que sigue siendo $copy(\kappa) + log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa))$.

---

iauxNuevoNodo(**in** c : κ, **in** v : α, **in** p : `puntero(nodo)`) $\rightarrow$ *res*: **puntero(nodo)**

```
var n : nodo
n.clave ← copy(c)                                                         // O(copy(κ))
n.valor ← v                                                               // O(1)
n.menor ← NULL                                                            // O(1)
n.mayor ← NULL                                                            // O(1)
n.padre ← p                                                               // O(1)
n.fdb ← 0                                                                 // O(1)
res ← &n                                                                  // O(1)
```

Complejidad: $\mathcal{O}(copy(\kappa))$

Justificación: 6*$\mathcal{O}(1)$ + $\mathcal{O}(copy(\kappa))$ = $\mathcal{O}(copy(\kappa))$

---

iauxDefinirNodo(**in/out** d: `diccLog(`κ, α`)`, **in/out** n: `puntero(nodo)`, **in** c: κ, **in** v: α)

if $(*n).clave = c$ **then** // $\mathcal{O}(equal(\kappa, \kappa))$
  $(*n).valor \leftarrow v$ // $\mathcal{O}(1)$
else
  if $c > (*n).clave$ **then** // $\mathcal{O}(greater(\kappa, \kappa))$
    if $(*n).mayor = NULL$ **then** // $\mathcal{O}(1)$
      $(*n).mayor \leftarrow auxNuevoNodo(c, v, n)$ // $\mathcal{O}(copy(\kappa))$
      $auxPropagarInsercion(d, (*n).mayor)$ // $\mathcal{O}(log(N))$
    else
      $auxDefinirNodo(d, (*n).mayor, c, v)$ // $\mathcal{O}(copy(\kappa) + (log(N) - 1)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$
    end if
  else
    if $(*n).menor = NULL$ **then**
      $(*n).menor \leftarrow auxNuevoNodo(c, v, n)$ // $\mathcal{O}(copy(\kappa))$
      $auxPropagarInsercion(d, (*n).menor)$ // $\mathcal{O}(log(N))$
    else
      $auxDefinirNodo(d, (*n).menor, c, v)$ // $\mathcal{O}(copy(\kappa) + (log(N) - 1)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$
    end if
  end if
end if

Complejidad: $\mathcal{O}(copy(\kappa) + log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$

Justificación: $log(N)$ veces $\mathcal{O}(equal(\kappa, \kappa)) + \mathcal{O}(greater(\kappa, \kappa)) + \mathcal{O}(1)$ para realizar la busqueda de donde insertar la nueva clave y $\mathcal{O}(copy(\kappa)) + \mathcal{O}(log(N))$ para crear el nodo y rebalancear el arbol.

$\mathcal{O}(copy(\kappa) + log(N)) + \mathcal{O}(log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$
$\mathcal{O}(copy(\kappa) + log(N) + log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$
$\mathcal{O}(copy(\kappa) + log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$

---

iauxPropagarInsercion(**in/out** d: `diccLog(`κ, α`)`, **in/out** n: `puntero(nodo)`)

if $(*n).fdb > 1 \;||\; (*n).fdb < -1$ **then** // $\mathcal{O}(1)$
  $auxBalancear(d, n)$ // $\mathcal{O}(1)$
else
  if $(*n).padre \neq NULL$ **then** // $\mathcal{O}(1)$
    if $n = (*(*n).padre).menor$ **then** // $\mathcal{O}(1)$
      $(*(*n).padre).fdb \leftarrow (*(*n).padre).fdb + 1$ // $\mathcal{O}(1)$
    else
      $(*(*n).padre).fdb \leftarrow (*(*n).padre).fdb - 1$ // $\mathcal{O}(1)$
    end if
    if $(*(*n).padre).fdb \neq 0$ **then** // $\mathcal{O}(1)$
      $auxPropagarBalance(d, (*n).padre)$ // $\mathcal{O}(log(N))$
    end if
  end if
end if

Complejidad: $\mathcal{O}(log(N))$

Justificación: Por cada llamada recursiva sube un nivel en el arbol. Como la cantidad de niveles es $log(N)$, donde N es la cantidad de elementos del arbol, en el peor de los casos hará $log(N)$ llamadas.

$\mathcal{O}(log(N)) + \mathcal{O}(1)*5 = \mathcal{O}(log(N))$

iauxBalancear(**in/out** d: `diccLog(`κ, α`)`, **in/out** n: `puntero(nodo)`)

if $(*n).fdb < 0$ **then** // $\mathcal{O}(1)$
  if $(*(*n).mayor).fdb > 0$ **then** // $\mathcal{O}(1)$
    $auxRotarDerecha(d, (*n).mayor)$ // $\mathcal{O}(1)$
  end if
  $auxRotarIzquierda(d, n)$ // $\mathcal{O}(1)$
else
  if $(*(*n).menor).fdb > 0$ **then** // $\mathcal{O}(1)$
    $auxRotarIzquierda(a, (*n).menor)$ // $\mathcal{O}(1)$
  end if
  $auxRotarDerecha(d, n)$ // $\mathcal{O}(1)$
end if

Complejidad: $\mathcal{O}(1)$

Justificación: Las rotaciones se realizan en $\mathcal{O}(1)$ y a lo sumo debe realizar 2 rotaciones para balancear el nodo.
$\mathcal{O}(1) = \mathcal{O}(1)*4$

iauxRotarIzquierda(**in/out** d: `diccLog(`κ, α`)`, **in/out** rr: `puntero(nodo)`)

var $nr : puntero(nodo)$
$nr \leftarrow (*rr).mayor$ // $\mathcal{O}(1)$
$(*rr).mayor \leftarrow (*nr).menor$ // $\mathcal{O}(1)$
if $(*nr).menor \neq NULL$ **then** // $\mathcal{O}(1)$
  $(*(*nr).menor).padre \leftarrow rr$ // $\mathcal{O}(1)$
end if
$(*nr).padre \leftarrow (*rr).padre$ // $\mathcal{O}(1)$
if $(*rr).padre = NULL$ **then** // $\mathcal{O}(1)$
  $d.raiz \leftarrow nr$ // $\mathcal{O}(1)$
else
  if $(*(*rr).padre).menor = rr$ **then** // $\mathcal{O}(1)$
    $(*(*rr).padre).menor \leftarrow nr$ // $\mathcal{O}(1)$
  else
    $(*(*rr).padre).mayor \leftarrow nr$ // $\mathcal{O}(1)$
  end if
end if
$(*nr).menor \leftarrow rr$ // $\mathcal{O}(1)$
$(*rr).parent \leftarrow nr$ // $\mathcal{O}(1)$
$(*rr).fdb \leftarrow (*rr).fdb + 1 - min((*nr).fdb, 0)$ // $\mathcal{O}(1)$
$(*nr).fdb \leftarrow (*nr).fdb + 1 + min((*rr).fdb, 0)$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Justificación: Se realiza una cantidad de operaciones constante en el peor caso sobre punteros de nodos y naturales.
$14*\mathcal{O}(1) = \mathcal{O}(1)$.

iauxRotarDerecha(**in/out** d: `diccLog(`κ, α`)`, **in/out** rr: `puntero(nodo)`)

var $nr : puntero(nodo)$
$nr \leftarrow (*rr).menor$ // $\mathcal{O}(1)$
$(*rr).menor \leftarrow (*nr).mayor$ // $\mathcal{O}(1)$
if $(*nr).mayor \neq NULL$ **then** // $\mathcal{O}(1)$
  $(*(*nr).mayor).padre \leftarrow rr$ // $\mathcal{O}(1)$
end if
$(*nr).padre \leftarrow (*rr).padre$ // $\mathcal{O}(1)$
if $(*rr).padre = NULL$ **then** // $\mathcal{O}(1)$
  $d.raiz \leftarrow nr$ // $\mathcal{O}(1)$
else
  if $(*(*rr).padre).menor = rr$ **then** // $\mathcal{O}(1)$
    $(*(*rr).padre).menor \leftarrow nr$ // $\mathcal{O}(1)$
  else
    $(*(*rr).padre).mayor \leftarrow nr$ // $\mathcal{O}(1)$
  end if
end if
$(*nr).mayor \leftarrow rr$ // $\mathcal{O}(1)$
$(*rr).parent \leftarrow nr$ // $\mathcal{O}(1)$
$(*rr).fdb \leftarrow (*rr).fdb - 1 + min((*nr).fdb, 0)$ // $\mathcal{O}(1)$
$(*nr).fdb \leftarrow (*nr).fdb - 1 - min((*rr).fdb, 0)$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

Justificación: Se realiza una cantidad de operaciones constante sobre punteros de nodos y naturales.
$14 * \mathcal{O}(1) = \mathcal{O}(1)$.

---

iBorrar(**in/out** d: `diccLog(`κ, α`)`, **in** c: κ)

$auxBorrarNodo(d, d.raiz, c)$ // $\mathcal{O}(log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$

Complejidad: $\mathcal{O}(log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$

Justificación: Asumiendo que la clave c esta definida puedo llamar a $auxBorrarNodo$. La complejidad es la misma que la del auxiliar.

iauxBorrarNodo(**in/out** d: `diccLog`(κ, α), **in** n: `puntero(nodo)`, **in** c: κ)

if $c = (*n).clave$ **then** // $\mathcal{O}(equal(\kappa, \kappa))$
  if $(*n).menor \neq NULL$ && $(*n).mayor \neq NULL$ **then** // $\mathcal{O}(1)$
    var $n2 : puntero(nodo)$
    $n2 \leftarrow auxMinimo((*n).mayor)$ // $\mathcal{O}(log(n))$
    $(*n).clave \leftarrow (*n2).clave$ // $\mathcal{O}(1)$
    $(*n).valor \leftarrow (*n2).valor$ // $\mathcal{O}(1)$
    $auxRecortar(d, n2)$ // $\mathcal{O}(log(N))$
  else
    $auxRecortar(d, n)$ // $\mathcal{O}(log(N))$
  end if
else
  if $c > (*n).clave$ **then** // $\mathcal{O}(greater(\kappa, \kappa))$
    $auxBorrarNodo(d, (*n).mayor, c)$ // $\mathcal{O}((log(N) - 1)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$
  else
    $auxBorrarNodo(d, (*n).menor, c)$ // $\mathcal{O}((log(N) - 1)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$
  end if
end if

Complejidad: $\mathcal{O}(log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$

Justificación: Debido a que se asume que la clave esta en el diccionario no hace falta chequear si $n = NULL$. En cada llamado recursivo a la funcion se desciende un nivel en el arbol, realizando comparaciones sobre la clave c. Al encontrar el nodo, se encarga de borrarlo en $\mathcal{O}(log(N))$. Como la cantidad de niveles del arbol es $log(N)$, donde N es la cantidad de elementos del mismo, realiza a lo sumo $log(N)$ comparaciones.

$\mathcal{O}(log(N)) + \mathcal{O}(log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$
$\mathcal{O}(log(N) + log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$
$\mathcal{O}(log(N)(greater(\kappa, \kappa) + equal(\kappa, \kappa)))$

---

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

if $(*n).menor = NULL$ **then** // $\mathcal{O}(1)$
  $res \leftarrow n$ // $\mathcal{O}(1)$
else
  $res \leftarrow auxMinimo((*n).menor)$ // $\mathcal{O}(log(N) - 1)$
end if

Complejidad: $\mathcal{O}(log(N))$

Justificación: En cada llamada baja un nivel del arbol buscando el elemento minimo. Como la cantidad de niveles del arbol es $log(N)$ donde N es la cantidad de elementos, en el peor caso debe descender por todo el arbol.

$\mathcal{O}(1) + \mathcal{O}(log(N)) = \mathcal{O}(log(N))$

iauxRecortar(**in/out** d: `diccLog(`κ, α`)`, **in** n: `puntero(nodo)`)

var $ho : puntero(nodo)$
if $(*n).menor \neq NULL$ **then** // $\mathcal{O}(1)$
  $ho \leftarrow (*n).menor$ // $\mathcal{O}(1)$
  $(*ho).padre \leftarrow (*n).padre$ // $\mathcal{O}(1)$
end if
if $(*n).mayor \neq NULL$ **then** // $\mathcal{O}(1)$
  $ho \leftarrow (*n).mayor$ // $\mathcal{O}(1)$
  $(*ho).padre \leftarrow (*n).padre$ // $\mathcal{O}(1)$
end if
if $(*n).padre = NULL$ **then** // $\mathcal{O}(1)$
  $d.raiz \leftarrow ho$ // $\mathcal{O}(1)$
else
  $auxPropagarBorrado(d, n)$ // $\mathcal{O}(log(N))$
  if $(*(*n).padre).menor = n$ **then** // $\mathcal{O}(1)$
    $(*(*n).padre).menor \leftarrow ho$ // $\mathcal{O}(1)$
  else
    $(*(*n).padre).mayor \leftarrow ho$ // $\mathcal{O}(1)$
  end if
end if

Complejidad: $\mathcal{O}(log(N))$

Justificación: Como no puede ocurrir el caso de que el nodo tenga dos hijos, a lo sumo solo se entra en uno de los primeros ifs.

. $\mathcal{O}(1)*8+ \mathcal{O}(log(N)) = \mathcal{O}(log(N))$

iauxPropagarBorrado(**in/out** d: `diccLog(`κ, α`)`, **in** n: `puntero(nodo)`)

if $(*n).fdb > 1 \;||\; (*n).fdb < -1$ **then** // $\mathcal{O}(1)$
  $auxBalancear(n)$ // $\mathcal{O}(1)$
end if
if $(*n).padre \neq NULL$ **then** // $\mathcal{O}(1)$
  if $(*(*n).padre).menor = n$ **then** // $\mathcal{O}(1)$
    $(*(*n).padre).fdb \leftarrow (*(*n).padre).fdb - 1$ // $\mathcal{O}(1)$
  else
    $(*(*n).padre).fdb \leftarrow (*(*n).padre).fdb + 1$ // $\mathcal{O}(1)$
  end if
  $auxPropagarBorrado(d, (*n).padre)$ // $\mathcal{O}(log(N))$
end if

Complejidad: $\mathcal{O}(log(N))$

Justificación: Por cada llamada recursiva sube un nivel en el arbol. Como la cantidad de niveles es $log(N)$, donde N es la cantidad de elementos del arbol, en el peor de los casos hará $log(N)$ llamadas.

$\mathcal{O}(log(N)) + \mathcal{O}(1)*5 = \mathcal{O}(log(N))$

6.4.1. Algoritmos del iterador

iCrearIt(**in** d: `diccLog(`κ, α`)`)

$res.dicc \leftarrow d$ // $\mathcal{O}(1)$
$res.actual \leftarrow d.raiz$ // $\mathcal{O}(1)$
$res.siguientes \leftarrow Vacia()$ // $\mathcal{O}(1)$
if $*res.actual).menor \neq NULL$ **then** // $\mathcal{O}(1)$
  $AgregarAdelante(res.siguientes, (*res.actual).menor)$ // $\mathcal{O}(copy(\alpha))$
end if
if $*res.actual).mayor \neq NULL$ **then** // $\mathcal{O}(1)$
  $AgregarAdelante(res.siguientes, (*res.actual).mayor)$ // $\mathcal{O}(copy(\alpha))$
end if

Complejidad: $\mathcal{O}(copy(\alpha))$
Justificación: $5*\mathcal{O}(1) + 2 * copy(\alpha) = \mathcal{O}(copy(\alpha))$

---

iHayMas?(**in** it: `itDiccLog(`κ, α`)`) $\rightarrow$ res: **bool**

$res \leftarrow it.actual \neq NULL$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iActual(**in** it: `itDiccLog(`κ, α`)`) $\rightarrow$ res: **tupla**$(clave : \kappa, significado : \alpha)$

$res \leftarrow <clave : (*it.actual).clave,\ significado : (*it.actual).valor>$ // $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

---

iAvanzar(**in** it: `itDiccLog(`κ, α`)`)

if $EsVacia?(res.siguientes)$ **then** // $\mathcal{O}(1)$
  $res.actual \leftarrow NULL$ // $\mathcal{O}(1)$
else
  $res.actual \leftarrow Primero(res.siguientes)$ // $\mathcal{O}(1)$
  $res.siguientes \leftarrow Fin(res.siguientes)$ // $\mathcal{O}(1)$
  if $*res.actual).menor \neq NULL$ **then** // $\mathcal{O}(1)$
    $AgregarAdelante(res.siguientes, (*res.actual).menor)$ // $\mathcal{O}(copy(\alpha))$
  end if
  if $*res.actual).mayor \neq NULL$ **then** // $\mathcal{O}(1)$
    $AgregarAdelante(res.siguientes, (*res.actual).mayor)$ // $\mathcal{O}(copy(\alpha))$
  end if
end if

Complejidad: $\mathcal{O}(copy(\alpha))$
Justificación: $6*\mathcal{O}(1) + 2 * copy(\alpha) = \mathcal{O}(copy(\alpha))$

6.5. Servicios usados

Se utilizan las funciones exportadas por el modulo Lista Enlazada(α). Se cuenta con que la complejidad de las operaciones Vacia, Primero, Fin y AgregarAdelante sea $\mathcal{O}(1)$.