Algoritmos y Estructuras de Datos II

Primer Cuatrimestre de 2016

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

Trabajo Práctico 2

Diseño

Grupo 18

Integrante	LU	Correo electrónico
Ignacio Manuel Fernandez	047/14	nachofernandez.1995@hotmail.com
Nicolas Ansaldi	128/14	nansaldi611@gmail.com
Nicolas Ippolito	724/14	ns_ippolito@hotmail.com
Facundo Pugliese	449/14	facu_pugliese@hotmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Algoritmos y Estructuras de Datos II

Primer Cuatrimestre de 2016

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

Trabajo Práctico 2

Diseño

Grupo 18

Integrante	LU	Correo electrónico
Ignacio Manuel Fernandez	047/14	nachofernandez.1995@hotmail.com
Nicolas Ansaldi	128/14	nansaldi611@gmail.com
Nicolas Ippolito	724/14	ns_ippolito@hotmail.com
Facundo Pugliese	449/14	facu_pugliese@hotmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Dato **4**
1.1 Especificación . . . 4
1.2 Interfaz . . . 4
1.3 Pautas de implementación . . . 5
1.3.1 Justificación . . . 5
1.3.2 Invariante de representación . . . 5
1.3.3 Predicado de abstracción . . . 5
1.4 Algoritmos . . . 5

2 Registro **8**
2.1 Especificación . . . 8
2.2 Interfaz . . . 8
2.3 Pautas de implementación . . . 10
2.3.1 Justificación . . . 10
2.3.2 Invariante de representación . . . 10
2.3.3 Predicado de abstracción . . . 10
2.4 Algoritmos . . . 10

3 Tabla **14**
3.1 Especificación . . . 14
3.2 Interfaz . . . 14
3.3 Pautas de implementación . . . 16
3.3.1 Justificación . . . 16
3.3.2 Invariante de representación . . . 16
3.3.3 Predicado de abstracción . . . 17
3.4 Algoritmos . . . 17

4 Base de Datos **29**
4.1 Especificación . . . 29
4.2 Interfaz . . . 29
4.3 Pautas de implementación . . . 30
4.3.1 Justificación . . . 30
4.3.2 Invariante de representación . . . 31
4.3.3 Predicado de abstracción . . . 31
4.4 Algoritmos . . . 31

5 DiccionarioLog(κ,σ) **37**
5.1 Especificación . . . 37
5.2 Pautas de implementación . . . 39
5.2.1 Justificación . . . 39
5.2.2 Invariante de representación . . . 39
5.2.3 Predicado de abstracción . . . 39
5.3 Algoritmos . . . 41

6 DiccionarioString(σ) **47**
6.1 Interfaz . . . 47
6.2 Pautas de implementacion . . . 49
6.2.1 Invariante de representación . . . 49
6.2.2 Predicado de abstracción . . . 49
6.3 Algoritmos . . . 51

1 Dato

1.1 Especificación

Se utiliza el *TAD DATO* especificado por la práctica

1.2 Interfaz

Interfaz

Género dato
se explica con: dato

Trabajo Práctico 2Operaciones de Dato DATOSTRING(**in** s: `string`) → res : `dato`
Pre ≡ {true}
Post ≡ { $res =_{\text{obs}}$ datoString(s) }
Complejidad: O(1)
Descripción: Creo un dato a partir de un string

DATONAT(**in** n: `nat`) → res : `dato`
Pre ≡ {true}
Post ≡ { $res =_{\text{obs}}$ datoNat(n) }
Complejidad: O(1)
Descripción: Creo un dato a partir de un nat

NAT?(**in** d: `dato`) → res : `bool`
Pre ≡ {true}
Post ≡ { $res =_{\text{obs}}$ Nat?(d)}
Complejidad: O(1)
Descripción: Devuelve true sii d es un nat

VALORNAT(**in** d: `dato`) → res : `nat`
Pre ≡ {Nat?(d)}
Post ≡ { $res =_{\text{obs}}$ valorNat(d) }
Complejidad: O(1)
Descripción: Devuelve el valor de un dato tipo nat

VALORSTR(**in** d: `dato`) → res : `string`
Pre ≡ {String?(d)}
Post ≡ { $res =_{\text{obs}}$ valorStr(d) }
Complejidad: O(1)
Descripción: Devuelve el valor de un tipo string

MISMOTIPO?(**in** d: `dato`, **in** d' : `dato`) → res : `bool`
Pre ≡ {true}
Post ≡ { $res =_{\text{obs}}$ mismoTipo?(d, d') }
Complejidad: O(1)
Descripción: Devuelve true sii d y d' son del mismo tipo

STRING?(**in** d: `dato`) → res : `bool`
Pre ≡ {true}
Post ≡ { $res =_{\text{obs}}$ String?(d) }
Complejidad: O(1)
Descripción: Devuelve true sii d es de tipo string

MIN(**in** cd: `conj(dato)`) → res : `dato`
Pre ≡ { $\neg\varnothing?(cd) \wedge_{\text{L}} (\forall d : dato) d \in cd \Rightarrow_{\text{L}}$mismoTipo?(dameUno(cd), d)}
Post ≡ { $res =_{\text{obs}}$ Min(cd)}
Complejidad: O(n), donde n es $\#(cd)$
Descripción: Devuelve el minimo valor (si hay varios devuelve el primero) del conjunto

MAX(**in** cd: `conj(dato)`) $\rightarrow$ res : `dato`
Pre $\equiv$ { $\neg\varnothing?(cd) \wedge_{\text{L}} (\forall d : dato) d \in cd \Rightarrow_{\text{L}}$ mismoTipo?(dameUno(cd), d) }
Post $\equiv$ { $res =_{\text{obs}}$ Max(cd)}
Complejidad: O(n), donde n es #(cd)
Descripción: Devuelve el maximo valor (si hay varios devuelve el primero) del conjunto

MENORIGUAL(**in** d_1 : `dato`, **in** d_2 : `dato`) $\rightarrow$ res : `bool`
Pre $\equiv$ {mismoTipo?(d_1, d_2)}
Post $\equiv$ { $res =_{\text{obs}} d_1 \leqslant d_2$ }
Complejidad: O(1)
Descripción: Devuelve true sii d_1 es menor o igual que d_2

1.3 Pautas de implementación

Representación

1.3.1 Justificación

tipo? indica que si es true el dato es de tipo nat, sino es string. Luego, con esta información accedo a alguno de los datos que interese y el otro es irrelevante.

`Dato` **se representa con** `superdt`

donde `superdt` es `tupla`(*tipo?*: `bool` ,
Nat: `nat` ,
String: `string`)

1.3.2 Invariante de representación

Informal

Siempre va a haber un dato para mirar por ende vale siempre

Formal

Rep : superdt s $\longrightarrow$ bool
Rep(s) $\equiv$ true $\Longleftrightarrow$ true

1.3.3 Predicado de abstracción

Abs : superdt s $\longrightarrow$ Dato {Rep(s)}
Abs(s) $\equiv$ $(\forall e : estr) Abs(e) =_{\text{obs}} d : dato \mid (e.tipo? = Nat?(d) \wedge_{\text{L}} Nat?(d) \Rightarrow_{\text{L}} valorNat(d) = e.Nat \vee String?(d) \Rightarrow_{\text{L}} valorStr(d) = e.String)$

1.4 Algoritmos

Algoritmos

Trabajo Práctico 2Algoritmos del módulo

---

iDatoString(**in** s : `string`) $\rightarrow$ res : superdt
1: $res \leftarrow < false, 0, s >$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Sólo realiza una asignación.

---

iDatoNat(**in** n : `nat`) $\rightarrow$ res : superdt

1: $res \leftarrow < true, n, vacia() >$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Sólo realiza una asignación. El vacio() es una string vacía.

iNat?(**in** s : `superdt`) $\rightarrow$ res : boolean

1: $res \leftarrow s.tipo?$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Sólo realiza una asignación.

iValorNat(**in** s : `superdt`) $\rightarrow$ res : nat

1: $res \leftarrow s.Nat$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Sólo realiza una asignación.

iValorStr(**in** s : `superdt`) $\rightarrow$ res : string

1: $res \leftarrow s.String$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Sólo realiza una asignación.

iMismoTipo?(**in** s : `superdt`, **in** s' : `superdt`) $\rightarrow$ res : boolean

1: $res \leftarrow s.tipo? = s'.tipo?$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Sólo realiza una asignación con una comparación.

iString?(**in** s : `superdt`) $\rightarrow$ res : boolean

1: $res \leftarrow \neg s.tipo?$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Sólo realiza una asignación.

iMin(**in** c : `conj(superdt)`) $\rightarrow$ res : superdt

1: $i \leftarrow CrearIt(c)$ ▷ $O(1)$
2: $res \leftarrow Siguiente(i)$ ▷ $O(1)$
3: **while** $HaySiguiente(i)$ **do** ▷ $O(n)$
4: $Avanzar(i)$ ▷ $O(1)$
5: **if** $iMenorOIgual(Siguiente(i), res)$ **then** $res \leftarrow Siguiente(i)$ **else** **fi** ▷ $O(1)$
6: **end while**

Complejidad: $O(n)$
Justificación: La función recorre todo el conjunto c.

iMax(**in** c: `conj(superdt)`) $\rightarrow$ res : superdt

1: $i \leftarrow CrearIt(c)$ ▷ $O(1)$
2: $res \leftarrow Siguiente(i)$ ▷ $O(1)$
3: **while** $HaySiguiente(i)$ **do** ▷ $O(n)$
4: $Avanzar(i)$ ▷ $O(1)$
5: **if** $\neg iMenorOIgual(Siguiente(i), res)$ **then** $res \leftarrow Siguiente(i)$ **else** **fi** ▷ $O(1)$
6: **end while**

Complejidad: $O(n)$
Justificación: La función recorre todo el conjunto c y la función menoroigual es O(1) ya que los strings son acotados.

iMenorOIgual(**in** s: `superdt`, **in** s': `superdt`) $\rightarrow$ res : boolean

1: **if** $s.tipo?$ **then**
2: $res \leftarrow s.Nat \leq s'.Nat$ ▷ $O(1)$
3: **else**
4: $i \leftarrow 0$ ▷ $O(1)$
5: **while** $i < Longitud(s.String) \wedge i < Longitud(s'.String) \wedge s.String_i = s'.String_i$ **do** ▷ $O(1)$
6: $i \leftarrow i + 1$ ▷ $O(1)$
7: **end while**
8: **if** $i = Longitud(s.String) \vee i = Longitud(s'.String)$ **then** ▷ $O(1)$
9: $res \leftarrow Longitud(e.String) \leq Longitud(s'.String)$ ▷ $O(1)$
10: **else**
11: $res \leftarrow ord(s.String) \leq ord(s'.String)$ ▷ $O(1)$
12: **end if**
13: **end if**

Complejidad: $O(1)$
Justificación: En el peor caso el ciclo tiene que recorrer todo un string s o s', pero estos strings son acotados por lo que la función es O(1).

2 Registro

2.1 Especificación

Se utiliza el *TAD REGISTRO* especificado por la práctica.

2.2 Interfaz

Interfaz

parámetros formales
géneros σ, κ
función COPIAR(**in** s: σ) $\rightarrow$ res : σ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} s$}
Complejidad: $O(copy(k))$
Descripción: función de copia de σ's

función $\bullet = \bullet$(**in** s_1, s_2: σ) $\rightarrow$ res : σ
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ $s_1 = s_2$}
Complejidad: $O(equal(s_1, s_2))$

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

se explica con: DICCIONARIO(κ,σ), ITERADOR BIDIRECCIONAL MODIFICABLE (TUPLA(κ,σ)).

géneros: `Diccionario Lineal(`κ`,`σ`)`, `itDicc(`κ`,`σ`)`

Trabajo Práctico 2Operaciones de Registro DEF?(**in** c: `campo`, **in** r: `registro`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {def?(c,r) $=_{\text{obs}}$ res}
Complejidad: O(1)
Descripción: devuelve true sii c esta definido en el registro.

OBTENER(**in** r: `registro` , **in** c: `campo`) $\rightarrow$ res : `dato`
Pre $\equiv$ {def?(c,r)}
Post $\equiv$ { alias($res =_{\text{obs}}$ obtener(c,r))}
Complejidad: O(1)
Descripción: Dada una clave obtengo su significado.
Aliasing: res es modificable si y solo si r es modificable

VACIO() $\rightarrow$ res : `registro`
Pre $\equiv$ {true}
Post $\equiv$ { $res =_{\text{obs}}$ vacio() }
Complejidad: O(1)
Descripción: Creo un registro vacio

DEFINIR(**in** c: `campo`, **in** d: `dato` , **in**/**out** r: `registro`) $\rightarrow$ res : `itDicc(campo,dato)`
Pre $\equiv$ { r $=_{\text{obs}}$ $r_0 \wedge \neg definido?(c, r)$}
Post $\equiv$ { r $=_{\text{obs}}$ definir(c,d,r_0) $\wedge$ haySiguiente(res) $\wedge_{\text{L}}$ Siguiente(res) = <c,d> $\wedge$ esPermutacion(SecuSuby(res),r)}
Complejidad: O(copy(d))
Descripción: Defino un campo con su dato en el registro.
Aliasing: Los elementos c y d se definen por copia. El iterador se invalida si y solo si se elimina el elemento siguiente del iterador sin utilizar la funcion ELIMINARSIGUIENTE. Además , anteriores(res) y siguientes(res) podrian cambiar completamente ante cualquier operacion que modifique r sin utilizar las funciones del iterador.

DefinirLento(**in** c: `campo`, **in** d: `dato` , **in/out** r: `registro`) → res : `itDicc(campo,dato)`
Pre ≡ { r =$_{obs}$ r_0 }
Post ≡ { r =$_{obs}$ definir(c,d,r_0) ∧ haySiguiente(res) ∧$_L$ Siguiente(res) = <c,d> ∧ esPermutacion(SecuSuby(res),r) }
Complejidad: O(copy(d))
Descripción: Defino un campo con su dato en el registro.
Aliasing: Los elementos c y d se definen por copia. El iterador se invalida si y solo si se elimina el elemento siguiente del iterador sin utilizar la funcion ELIMINARSIGUIENTE. Además , anteriores(res) y siguientes(res) podrian cambiar completamente ante cualquier operacion que modifique r sin utilizar las funciones del iterador.

Borrar(**in** c: `campo` , **in/out** r: `registro`)
Pre ≡ { r =$_{obs}$ r_0 ∧ def?(c,r_0) }
Post ≡ { r =$_{obs}$ borrar(c,r_0) }
Complejidad: O(1)
Descripción: Borra un campo del registro.

campos(**in** r: `registro`) → res : `conj(campos)`
Pre ≡ {true}
Post ≡ { res =$_{obs}$ campos(r)}
Complejidad: O(1)
Descripción: Devuelve el conjunto de campos del registro

borrar?(**in** $crit$: `registro`, **in** reg: `registro`) → res : `bool`
Pre ≡ { $\#campos(crit) = 1$ }
Post ≡ { res =$_{obs}$ borrar?(crit,reg)}
Complejidad: O(1)
Descripción: Devuelve true sii puedo borrar un registro según el criterio

agregarCampos(**in** r_1: `registro`, **in** r_2: `registro`) → res : `registro`
Pre ≡ { r_1 =$_{obs}$ $r1_0$ }
Post ≡ { res =$_{obs}$ agregarCampos(reg_0, r_2)}
Complejidad: $O(\#(Campos(r_1) - Campos(r_2)) * L$
Descripción: Agrega los campos del registro r2 (junto con su respectivo dato) al registro r1

copiarCampos(**in** cc: `conj(campos)`, **in/out** r_1: `registro`, **in** r_2: `registro`) → res : `registro`
Pre ≡ {r_1 =$_{obs}$ r_0 ∧ cc ⊆ campos(r_2)}
Post ≡ { r_1 =$_{obs}$ copiarCampos(cc,r_0,r_2) }
Complejidad: $O(\#cc) * L$
Descripción: Devuelve un registro res con los campos de r_1 junto con los campos de r_2, con sus valores, y si hay un campo en comun sobreescribe el valor de r_1 por el de r_2 (en el registro que devuelve)

coincideAlguno(**in** r_1: `registro`, **in** cc: `conj(campo)`, **in** r_2: `registro`) → res : `bool`
Pre ≡ { $cc \subseteq$ (campos(r_1) ∩ campos(r_2)) }
Post ≡ { res =$_{obs}$ coincideAlguno(r_1,cc,r_2) }
Complejidad: O($(\#Claves(r_1) * \#Claves(r_2))$)
Descripción: Indica si hay algun campo en cc tal que en r1 y r2 su dato en ese campo sea el mismo.

coincidenTodos(**in** r_1: `registro`, **in** cc: `conj(campo)`, **in** r_2: `registro`) → res : `bool`
Pre ≡ { $cc \subseteq$ (campos(r_1) ∩ campos(r_2)) }
Post ≡ { res =$_{obs}$ coincidenTodos(r_1,cc,r_2) }
Complejidad: O($(\#Claves(r_1) * \#Claves(r_2))$)
Descripción: Indico si todos los campos de cc tienen el mismo dato en r_1 y r_2

enTodos(**in** c: `campo`, **in** cr: `conj(registro)`) → res : `bool`
Pre ≡ { }
Post ≡ { res =$_{obs}$ $enTodos(c, cr)$}
Complejidad: O(??)
Descripción: Indico si el campo c está en todos los registros del conjunto cr

combinarTodos(**in** c: `campo`, **in** r_1: `registro`, **in** cr: `conj(registro)`)
Pre ≡ { $c \in campos(r_1) \wedge enTodos(c, cr)$ }

Post $\equiv$ { $res =_{\text{obs}} combinarTodos(c, r_1, cr)$ }
Complejidad: O(??)
Descripción: Agrega al registro r_1

DiferenciaSimetrica(**in** r_1 : `registro`, **in** r_2 : `registro`) $\rightarrow$ res : `conj(campo)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ campos(r_1)-campos(r_2)}
Complejidad: O(1)
Descripción: Devuelve el conjuntu de campos formado por los campos que estan en r_2 pero no en r_1

2.3 Pautas de implementación

Representación

2.3.1 Justificación

El TAD Registro extiende a Diccionario(campo,dato), y creemos que al representarlo con un dicc(campo,dato) podemos asegurar las complejidades que debemos respetar para hacer los modulos posteriores.

`Registro` **se representa con** `dicc(campo,dato)`

2.3.2 Invariante de representación

Rep : dicc(campo $\times$ dato) d $\longrightarrow$ bool
Rep(d) $\equiv$ true $\Longleftrightarrow$ true

2.3.3 Predicado de abstracción

Abs : dicc(campo $\times$ dato) d $\longrightarrow$ Registro {Rep(d)}
Abs(d) $\equiv$ $(\forall d : dicc(campo, dato))Abs(e) =_{\text{obs}} r : Registro \mid (\forall c : campo)def?(c, r) = def?(c, d) \wedge (\forall t : dato)obtener(c, t, r) = obtener(c, t, d)$

2.4 Algoritmos

Algoritmos

Trabajo Práctico 2Algoritmos del módulo

---

iVacio() $\rightarrow$ res : registro
1: $res \leftarrow vacio()$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Usa la operacion Vacio() del diccionario lineal de los apuntes

---

iDef?(**in** c: `campo`, **in** r : `dicc(campo,dato)`) $\rightarrow$ res : $bool$
1: $res \leftarrow Definido?(r, c)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Usa la operacion Definido? del diccionario lineal de los apuntes

---

iObtener(**in** c: `campo`, **in** r: `dicc(campo,dato)`) $\rightarrow$ res : $dato$

1: $res \leftarrow Significado(r, c)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Usa la operacion Significado del diccionario lineal de los apuntes. Normalmente tiene complejidad lineal pero como la cantidad de campos de un registro está acotada, entonces es O(1).

iDefinir(**in** c: `campo`, **in** d: `dato`, **in** r: `dicc(campo,dato)`)

1: $DefinirRapido(r, c, d)$ ▷ $O(copy(c) + copy(d))$

Complejidad: $O(L)$
Justificación: Usa la operacion DefinirRapido del diccionario lineal de los apuntes. El copy(c) es O(1) ya que campo es un string acotado y la complejidad de copiar un nat es O(1).

iDefinirLento(**in** c: `campo`, **in** d: `dato`, **in** r: `dicc(campo,dato)`)

1: $Definir(r, c, d)$ ▷ $O(copy(c) + copy(d))$

Complejidad: $O(L)$
Justificación: Usa la operacion Definir del diccionario lineal de los apuntes. Normalmente la complejidad seria lineal sumada a la complejidad de copiar y comparar las claves y copiar el significado, pero como los campos son acotados la comparacion y el copiado termina siendo O(1), mientras que la complejidad de copiar el significado termina siendo O(L), done L es el string mas largo.

iBorrar(**in** r: `dicc(campo,dato)`, **in** c: `campo`)

1: $Borrar(r, c)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Usa la operacion Borrar del diccionario lineal de los apuntes

iCampos(**in** r: `dicc(campo,dato)`) $\rightarrow$ res : $conj(campo)$

1: $res \leftarrow vacio()$ ▷ $O(1)$
2: $i \leftarrow CrearIt(r)$ ▷ $O(1)$
3: **while** $HaySiguiente(i)$ **do** ▷ $O(\#CLAVES(r))$
4: $Agregar(res, SiguienteClave(i))$ ▷ $O(1)$
5: $Avanzar(i)$ ▷ $O(1)$
6: **end while**

Complejidad: $O(1)$
Justificación: Este algoritmo usa el iterador de diccionario lineal del apunte de modulos basicos de la catedra. Se usa conjunto lineal para represenar res y la operacion Agregar es del conjunto lineal del apunte de modulos basicos. # Claves(r) es la operacion del modulo basico de diccionario lineal que calcula la cantidad de claves del registro. Es O(1) ya que la cantidad de campos de un registro es acotada

iBorrar?(**in** $crit$: `dicc(campo,dato)`, **in** reg: `dicc(campo,dato)`) $\rightarrow$ res : $bool$

1: $res \leftarrow iCoindicenTodos(crit, iCampos(crit), reg)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: El algoritmo sólo realiza una asignación (y coincidentodos es O(1))

iAgregarCampos(**in** r_1 : `dicc(campo,dato)`, **in** r_2 : `dicc(campo,dato)`) $\rightarrow$ res : $dicc(campo, dato)$

1: $res \leftarrow CopiarCampos(DiferenciaSimetrica(r_1, r_2), r_1, r_2)$ ▷ O($\#(cc) * copy(Obtener(Siguiente(it)))$

Complejidad: O($\#(cc) * copy(Obtener(Siguiente(it)))$
Justificación: Este algoritmo utiliza una sola vez la función copiarCampos y por ende su complejidad es la misma que la de esa función.

iCopiarCampos(**in** cc: `conj(campo)`, **in** r_1: `dicc(campo,dato)`, **in** r_2: `dicc(campo,dato)`) $\rightarrow$ res : $conj(dicc(campo, dato))$

1: Conj(campo) $it \leftarrow CrearIt(cc)$ ▷ $O(1)$
2: $res \leftarrow Copiar(r_1)$ ▷ $O(L)$
3: **while** $HaySiguiente(it)$ **do** ▷ $O(\#(cc) * L)$
4: **if** $iDef?(Siguiente(it), r_1)$ **then**
5: $DefinirLento(Siguiente(it), Obtener(Siguiente(it), r_2), res)$ ▷ $O(L)$
6: **else**
7: $Definir(Siguiente(it), Obtener(Siguiente(it), r_2), res)$ ▷ $O(L)$
8: **end if**
9: $Avanzar(it)$ ▷ $O(1)$)
10: **end while**

Complejidad: $O(\#(cc) * L)$
Justificación: Este algoritmo itera por el conjunto de campos y en cada iteración agrega el campo con su respectivo dato a r1

iCoincideAlguno(**in** r_1 : `dicc(campo,dato)`, **in** cc: `conj(campo)`, **in** r_2 : `dicc(campo,dato)` $\rightarrow$ res : $bool$

1: $b \leftarrow false$ ▷ $O(1)$
2: $it_1 \leftarrow CrearIt(r_1)$ ▷ $O(1)$
3: **while** $(HaySiguiente(it_1) \wedge \neg b)$ **do** ▷ $O(\#(cc))$
4: **if** $(obtener(Siguiente(it_1), r_1) = obtener(Siguiente(it_1), r_2))$ **then** ▷ $O(1)$
5: $b \leftarrow true$ ▷ $O(1)$
6: **end if**
7: $Avanzar(it_1)$ ▷ $O(1)$
8: **end while**
9: $res \leftarrow b$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Esto es O(1) ya que, como la cantidad de campos de un registro está acotada, entonces la cantidad de campos del registro cc va a estar acotada ya que por precondición está contenida en los campos de r1 y los campos de r2

iCoincidenTodos(**in** r_1 : `dicc(campo,dato)`, **in** cc: `conj(campo)`, **in** r_2 : `dicc(campo,dato)` $\rightarrow$ res : $bool$

```
1: b ← true                                                          ▷ O(1)
2: it_1 ← CrearIt(cc)                                                ▷ O(1)
3: while (HaySiguiente(it_1) ∧ b) do                                 ▷ O(#(cc))
4:     if (obtener(Siguiente(it_1), r_1) ≠ obtener(Siguiente(it_1), r_2)) then   ▷ O(1)
5:         b ← false                                                 ▷ O(1)
6:     end if
7:     Avanzar(it_1)                                                 ▷ O(1)
8: end while
9: res ← b                                                           ▷ O(1)
```

Complejidad: $O(1)$
Justificación: Esto es O(1) ya que, como la cantidad de campos de un registro está acotada, entonces la cantidad de campos del registro cc va a estar acotada ya que por precondición está contenida en los campos de r1 y los campos de r2

---

iDiferenciaSimetrica(**in** r_1 : `Dicc(campo, dato)`, **in** r_2 : `Dicc(campo, dato)`) $\rightarrow$ res : $conj(campo)$

```
1: res ← Vacio()
2: ItConj(campo) it ← CrearIt(campos(r_2))
3: while HaySiguiente(it) do                                         ▷ O(#(campos(r_2)))
4:     if ¬(Pertenece?(Campos(r_1), Siguiente(i))) then
5:         Agregar(res, Siguiente(it))                               ▷ O(1)
6:     end if
7:     Avanzar(it)                                                   ▷ O(1)
8: end while
```

Complejidad: $O(1)$
Justificación: En este caso como los campos en los registros son acotados, y estos a la vez tambien lo son, recorrerlos y compararlos toman complejidad constante, por lo que la cantidad de campos de r_2, en complejidad de peor caso, termina siendo O(1)

3 Tabla

3.1 Especificación

Se utiliza el *TAD TABLA* especificado por la práctica.

3.2 Interfaz

Género tabla
se explica con: Registro , Campo , Dato, Conjunto (σ) , Secuencia(σ)

NuevaTabla(**in** *nombre*: `string`, **in** *claves*: `conj(campo)`, **in** *columnas*: `registro`) $\rightarrow$ *res* : `tabla`
Pre $\equiv$ {$\neg\varnothing(claves) \wedge claves \subseteq campos(columnas)$}
Post $\equiv$ { $res =_{\text{obs}} nuevaTabla(nombre, claves, columnas)$ }
Complejidad: O(1)
Descripción: Creo una nueva tabla

AgregarRegistro(**in** r: `registro`, **in/out** t: `tabla`)
Pre $\equiv$ {t $=_{\text{obs}}$ t_o $\wedge$ campos(t) $=_{\text{obs}}$ campos(r) $\wedge$ puedoInsertar?(r,t)}
Post $\equiv$ { t $=_{\text{obs}}$ agregarRegistro(r, t_o) }
Complejidad: O(L+log(n)), en caso promedio, donde n es la cantidad de registros de la tabla y L es el string mas largo
Descripción: Agrega un registro a la tabla

BorrarRegistro(**in** *criterio*: `registro`, **in/out** t: `tabla`)
Pre $\equiv$ {t $=_{\text{obs}}$ t_o $\wedge$ $\sharp$(campos(criterio)) $= 1$ $\wedge_{\text{L}}$ dameUno(campos(criterio) $\in$ claves(t)}
Post $\equiv$ {t $=_{\text{obs}}$ borrarRegistro(criterio, t) }
Complejidad: O(L + in), siendo in log(n) en caso promedio
Descripción: Borra un solo registro de la tabla segun el criterio, ya que el campo del registro criterio es un campo clave de la tabla

Indexar(**in** c: `campo`, **in/out** t: `tabla`)
Pre $\equiv$ {puedeIndexar(c,t) $\wedge$ t $=_{\text{obs}}$ t_o}
Post $\equiv$ {t $=_{\text{obs}}$ indexar(c, t_o)}
Complejidad: $O(n^2)$
Descripción: Agrega índices al campo c

TipoCampo(**in** c: `campo`, **in** t: `tabla`) $\rightarrow$ *res* : `tipo`
Pre $\equiv$ {c $\in$ campos(t)}
Post $\equiv$ {res $=_{\text{obs}}$ tipoCampo(c,m)}
Complejidad: O(1)
Descripción: Devuelve el tipo del campo, si es true el campo es nat sino string

Registros(**in** t: `tabla`) $\rightarrow$ *res* : `conj(registro)`
Pre $\equiv$ {true}
Post $\equiv$ { $res =_{\text{obs}}$ registro(t) }
Complejidad: O(1)
Descripción: Devuelve los registros de la tabla

Claves(**in** t: `tabla`) $\rightarrow$ *res* : `conj(campo)`
Pre $\equiv$ {true}
Post $\equiv$ { $res =_{\text{obs}}$ claves(t) }
Complejidad: O(1)
Descripción: Devuelve los campos clave de la tabla

Minimo(**in** c: `campo`, **in** t: `tabla`) $\rightarrow$ *res* : `dato`
Pre $\equiv$ {$\neg\varnothing$?(registros(t) $\wedge$ c $\in$ indices(t)}
Post $\equiv$ {res $=_{\text{obs}}$ mínimo(c,t)}

Complejidad: O(1)
Descripción: Devuelve el mínimo de un campo indexado

MAXIMO(**in** c: `campo`, **in** t: `tabla`) → res : `dato`
Pre ≡ {¬∅?(registros(t) ∧ c ∈ indices(t)}
Post ≡ {res =$_{obs}$ máximo(c,t)}
Complejidad: O(1)
Descripción: Devuelve el máximo de un campo indexado

INDICES(**in** t: `tabla`) → res : `conj(campo)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ indices(t)}
Complejidad: O(1)
Descripción: Devuelve los campos indexados de la tabla

COMPATIBLE(**in** r: `registro`,**in** t: `tabla`) → res : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ compatible(r, t)}
Complejidad: O(1)
Descripción: Devuelve true sii los campos de los registros coinciden con los de la tabla y tiene el mismo tipo

COMBINARREGISTROS(**in** c: `campo`,**in** t_1: `tabla`, **in** t_2: `tabla`) → res : `conj(registro`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ combinarRegistro(c, registros($_1$), registros(cr_2))}
Complejidad: O((n+m)*(L+log(n+m))) donde n y m son la cantidad de registros de t_1 y t_2 y L es la longitud del string más largo definido
Descripción: Devuelve el conjunto de registros formados a partir de que coincidan los datos de los registros de ambas tablas en el campo c, y uniendo los campos priorizando los valores de los registros de t_1

DAMECOLUMNA(**in** c: `campo`,**in** cr: `conj(registro)`) → res : `conj(dato)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ dameColumna(c, cr)}
Complejidad: $O(n^2)$
Descripción: $Devuelvelosvalores, sinrepetidos, delacolumna$

MISMOSTIPOS(**in** r: `registro`,**in** t: `tabla`) → res : `bool)`
Pre ≡ {campos(r) ⊆ campos(t)}
Post ≡ {res =$_{obs}$ mismosTipos(r, t)}
Complejidad: O(1)
Descripción: Devuelve true sii los campos del registro tienen los mismos tipos que los campos de la tabla

BUSCART(**in** t: `tabla`,**in** r: `registro`) → res : `conj(registro)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ coincidencias(r, registros(mt))}
Complejidad: O(L + log(n))
Descripción: Devuelve el conjunto de registro de la tabla que coinciden con el pasado por parametro

CANTIDADDEACCESOS(**in** t: `tabla`) → res : `nat`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ cantidadDeAccesos(t)}
Complejidad: O(1)
Descripción: Devuelve los registros de una tabla

ESTA(**in** r: `registro`, **in** t: `tabla`) → res : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ Esta(r,t)}
Complejidad: $\Theta(log(n) + L)$
Descripción: res es true sii el registro esta en la tabla

BuscaryBorrar(**in** *crit*: `registro`, **in/out** *mt*: `tabla`)
Pre ≡ {mt =$_{\text{obs}}$ mt_o}
Post ≡ {mt =$_{\text{obs}}$ borrarRegistro(crit, mt_o)}
Complejidad: $O(n * L)$

3.3 Pautas de implementación

Representación

3.3.1 Justificación

megatab representa la Tabla de la especificacion de Base de datos. La componente modificaciones indica la cantidad de veces que se agregaron y borraron registros en una tabla.IndicesUsados es un tupla que indica si hay un campo indexado tanto en un campo de tipo nat o de tipo string.

indiceN representa el campo indexado que es de tipo nat , y ademas guarda el minimo y maximo dato. Como los datos en este campo se distribuyen de forma uniforme elegimos regpordato (que es un diccLOG) para acceder a los datos (que son las claves de diccLOG) y sus respectivos significados son una lista de iteradores a los registros donde aparece el dato, además de un iterador que apunta al significado de su contraparte indiceS que apunta al mismo registro (si este indice está definido). En el caso de que el campo sea clave esta lista solo tiene un elemento. Esto permite actualizar bien los registros que tienen datos string (en campos indexados) ya que es posible acceder al regpordato en indiceS que este ultimo accede a los registros de la tabla.

indiceS representa el campo indexado de tipo string. Ademas se guarda el maximo y minimo dato del campo. En regpordato (que es un diccString) las claves son los datos del campo y sus respectivos significados son una lista de iteradores a los registros donde aparece el dato, además de un iterador que apunta al significado de su contraparte indiceN que apunta al mismo registro (si este indice está definido). En el caso donde el campo sea clave la lista solo tiene un elemento. Esto permite actualizar bien los registros donde tienen datos nat (en campos indexados) ya que puedo acceder al regpordato en indiceN que este ultimo accede a los registros de la tabla.

Nombre es el nombre de la tabla, claves es el conjunto de campos que son claves. Columnas es un registro que representa todos los campos de la tabla, permitiendo así también saber el tipo (Nat o String) asociado a cada campo. Registros representa una lista con todos los registros de la tabla.

`Tabla` **se representa con** `megatab`

donde `megatab` es `tupla`(*modificaciones*: `nat` ,
indiceN: `indiceNat` ,
indiceS: `indiceStr` ,
indicesUsados: `tupla <Nat: Bool, String: Bool>` ,
nombre: `String` ,
claves: `conj(String)` ,
columnas: `registro` ,
registros: `lista(registro)`)

donde `indiceNat` es `tupla`(*campo*: `String` ,
mínimo: `dato` ,
máximo: `dato` ,
regpordato: `diccLOG(dato, lista(apuntador))`)

donde `indiceStr` es `tupla`(*campo*: `String` ,
mínimo: `dato` ,
máximo: `dato` ,
regpordato: `diccString(dato, lista(apuntador))`)

donde `apuntador` es `tupla`(*reg*: `itLista(registro)` ,
compadre: `itLista(apuntador)`)

3.3.2 Invariante de representación

Informal

(1) *mt.claves* es un subconjunto de *mt.columnas*
(2) Si el primer elemento de la tupla *mt.usados* es *true*, entonces existe un campo *c* que pertenece a *mt.columnas* donde:
2.1 - El primer elemento de la tupla *mt.indiceN* es igual a *c*
2.2 - Las claves del cuarto elemento de la tupla *mt.indiceN* son de tipo *Nat*
2.3 - Toda clave del cuarto elemento de *mt.indiceN* tiene un significado distinto de vacío
2.4 - El segundo y tercer elemento de *mt.indiceN* son de tipo *Nat* y pertenecen a las claves del cuarto elemento de *mt.indiceN*
2.5 - Todas las listas unidas de *reg* en los significados de *regpordato* en *indiceN* apuntan una permutacion de los registros de mt.registros
(3) Si el segundo elemento de *mt.indiceUsado* es *true* entonces existe un nombre de campo *c'* con *c* distinto a *c'* y *c'* perteneciente a *mt.columnas* donde:
3.1 - El primer elemento de *mt.indiceS* es *c'*
3.2 - Las claves del cuarto elemento de *mt.indiceS* son del tipo *string*
3.3 - Todas las claves del cuarto elemento de *mt.indiceS* tienen un significado distinto de vacío
3.4 - El segundo y tercer elemento de *mt.indiceS* son de tipo *string* y pertenecen a las claves del cuarto elemento de *mt.indiceS*
3.5 - Todas las listas unidas de *reg* en los significados de *regpordato* en *indiceS* apuntan a una permutacion de los registros de mt.registros
(4) Si la primer componente y la segunda componente de *mt.indiceUsado* es *true* entonces :
4.1 - En *mt.indiceN* , para cada clave de la componente *mt.regpordato* su significado en la componente *compadre* apunta al signifcado de la clave correspondiente en *mt.indiceS.regpordato*
4.2 - En *mt.indiceS* , para cada clave de la componente *mt.regpordato* su significado en la componente *compadre* apunta al signifcado de la clave correspondiente en *mt.indiceN.regpordato*

Formal

Rep : megatabla $mt \longrightarrow$ bool
Rep(mt) $\equiv$ true $\iff$ 1) $mt.claves \subseteq mt.columnas \wedge$
2) $(\Pi_1(mt.indiceUsado) = true) \Rightarrow_{\text{L}} ((\forall c : campo)c \in mt.columnas \wedge$
2.1) $\Pi_1(mt.indiceN) = c \Rightarrow_{\text{L}}$
2.2) $((\forall d : dato)d \in claves(\Pi_1(mt.indiceN)) \Rightarrow (Nat?(d) = true) \wedge_{\text{L}}$
2.3) $\neg\varnothing?(obtener(d, \Pi_4(mt.indiceN)))) \wedge_{\text{L}}$
2.4) $\Pi_2(mt.indiceN) \in claves(\Pi_4(mt.indiceN)) \wedge \Pi_3(mt.indiceN) \in claves(\Pi_4(mt.indiceN)) \wedge_{\text{L}}$
3) $(\Pi_2(mt.indiceUsado) = true) \Rightarrow_{\text{L}}$
3.1) $((\forall c : campo)c \in mt.columnas \wedge \Pi_1(mt.indiceS) = c \Rightarrow_{\text{L}} ((\forall d : dato)d \in claves(\Pi_1(mt.indiceS))$
$\Rightarrow$
3.2) $(\neg Nat?(d)) \wedge$
3.3) $\neg\varnothing?(obtener(d, \Pi_4(mt.indiceS))) \wedge_{\text{L}}$
3.4) $\Pi_2(mt.indiceS) \in claves(\Pi_4(mt.indiceS)) \wedge \Pi_3(mt.indiceS) \in claves(\Pi_4(mt.indiceS))$

3.3.3 Predicado de abstracción

Abs: megatabla mt $\rightarrow$ Tabla
Abs(mt) = t : Tabla | nombre(t) $=_{\text{obs}}$ *mt.nombre* $\wedge$ $columnas(t) =_{\text{obs}}$ *mt.columnas* $\wedge$ $claves(t) =_{\text{obs}}$ *mt.claves* $\wedge$ $registros(t) =_{\text{obs}}$ *mt.registros* $\wedge$ $mt.modificaciones =_{\text{obs}} cantidadDeAccesos(t) \wedge \Pi_1(mt.indiceUsado) = true \Rightarrow_{\text{L}}$ $\Pi_1(mt.indiceN) \in indices(t) \wedge \Pi_2(mt.indiceUsado) = true \Rightarrow_{\text{L}} \Pi_1(mt.indiceS) \in indices(t) \wedge$

3.4 Algoritmos

Algoritmos

Trabajo Práctico 2Algoritmos del módulo

iNuevaTabla(**in** s : `string`, **in** *claves* : `conj(campos)`, **in** *columnas* : `registro`) → *res* : *megatab*

1: string c ← Default ▷ O(1)
2: int m ← 0 ▷ O(1)
3: string str ← $Vacio()$ ▷ O(1)
4: $res \leftarrow < 0, < c, m, m, Vacio() >, < c, str, str, Vacio() >, < false, false >, s, claves, columnas, Vacio() >$ ▷ $O(1)$

Complejidad: $O(1)$)
Justificación: Se crea una nueva tabla vacia definiendo los campos, designando cuales son clave, y el nombre de la tabla. Designo valores default para los indices indexados porque no esta acitvado el flag IndicesUsados, una vez que se activa se guardan valores que correspondan con la tabla.El Vacio es el del diccLOG y diccString para indiceN y indiceS respectivamente.

iAgregarRegistro(**in** r : `registro`, **in**/**out** mt : `megatab`)

1: $AgregarAtras(mt.registros, r)$ ▷ Copiar un registro es lineal en la longitud del string más largo (campos acotados) // $O(L)$
2: $itLista(registro)\ it \leftarrow CrearItUlt(mt.registros)$ ▷ $O(1)$
3: $Retroceder(it)$ ▷ Muevo al iterador para que tenga al nuevo registro como siguiente // $O(1)$
4: **if** $mt.IndicesUsados.Nat$ **then** ▷ $O(log(n))$
5: $int\ n \leftarrow Obtener(mt.incideN.campo, r)$
6: **if** $\neg(MenorOIgual(mt.indiceN.minimo, n))$ **then** ▷ $O(log(n))$
7: $mt.indiceN.minimo \leftarrow n$ ▷ $O(log(n))$
8: **end if**
9: **if** $MenorOIgual(mt.indiceN.maximo, n)$ **then** ▷ $O(log(n))$
10: $mt.IndiceN.maximo \leftarrow n$ ▷ $O(log(n))$
11: **end if**
12: **if** $Definido?(mt.indiceN.regpordato, n)$ **then** ▷ $O(log(n))$
13: $AgregarAtras(Obtener(mt.indiceN.regpordato, n), < it, CrearIt(Vacia()) >)$ ▷ $O(log(n)) + O(1) = O(log(n))$
14: **else**
15: $Definir(mt.indiceN.regpordato, n, AgregarAtras(Vacia(), < it, CrearIt(Vacia()) >))$ ▷ $O(log(n)) + O(1) = O(log(n))$
16: **end if**
17: **if** $mt.IndicesUsados.String$ **then**
18: $itLista(apuntador)\ In \leftarrow Retroceder(CrearItUlt(Obtener(mt.indiceN.regpordato, n)))$ ▷ $O(1)$
19: **end if**
20: **end if**
21: **if** $mt.IndicesUsados.String$ **then** ▷ $O(L)$
22: $string\ s \leftarrow Obtener(mt.incideS.campo, r)$ ▷ $O(L)$
23: **if** $\neg(MenorOIgual(mt.IndiceS.Minimo, s))$ **then** ▷ $O(L)$
24: $mt.IndiceS.minimo \leftarrow s$ ▷ $O(L)$
25: **end if**
26: **if** $MenorOIgual(mt.indiceS.maximo, s)$ **then** ▷ $O(L) + O(L) = O(L)$
27: $mt.indiceS.maximo \leftarrow s$ ▷ $O(L)$
28: **end if**
29: **if** $Definido?(mt.indiceS.regpordato, s)$ **then** ▷ $O(L) + O(L) = O(L)$
30: $AgregarAtras(Obtener(mt.indiceS.regpordato, s), < it, CrearIt(Vacio()) >)$ ▷ $O(L)$
31: **else**
32: $Definir(mt.indiceS.regpordato, s, AgregarAtras(Vacio(), < it, CrearIt(Vacio()) >))$ ▷ $O(L) + O(L) + O(1) = O(L)$
33: **end if**
34: **if** $mt.IndicesUsados.Nat$ **then**
35: $ItLista(apuntador)\ Is \leftarrow Retroceder(CrearItUlt(Obtener(mt.indiceS.regpordato, s)))$ ▷ $O(1)$
36: **end if**
37: **end if**
38: **if** $(mt.indicesUsados.Nat \wedge mt.indicesUsados.Str)$ **then** ▷ Cruzo las referencias // $O(1)$
39: $Siguiente(Is).compadre \leftarrow In$ ▷ $O(1)$
40: $Siguiente(In).compadre \leftarrow Is$ ▷ $O(1)$
41: **end if**
42: $mt.modificaciones \leftarrow mt.modificaciones + 1$ ▷ $O(1)$

Complejidad: $O(L + in)$
Justificación: Si no hay un indice de tipo Nat definido, el algoritmo no ejecuta el if de las lineas 4-20 y por lo tanto su complejidad es $O(1)$. Se ejecute o no el if de las lineas 11-37, la complejidad es lineal la longitud del string más largo (L) por el costo de copiado de la línea 1.

iBorrarRegistro(**in/out** t: `megatab`, **in** $crit$: `registro`))

```
1:  t.modificaciones ← t.modificaciones + 1                                                          ▷ O(1)
2:  itDicc(campo, dato) j ← CrearIt(crit)                                                            ▷ O(1)
3:  campo c ← SiguienteClave(j)                                                                      ▷ Pasado por referencia O(1)
4:  dato d ← SiguienteSignificado(j)                                                                 ▷ O(1)
5:  if (Nat?(d) = tipoCampo(c, t)) then                                                              ▷ O(1)
6:      if (t.indicesUsados.Nat ∧ t.indiceN.campo = c) then                                          ▷ O(1)
7:          it ← Buscar(t.indiceN.regpordato, d)                                                     ▷ O(log(n))
8:          if (SiguienteClave(it) = d) then                                                         ▷ O(1)
9:              if (t.indicesUsados.String) then                                                     ▷ O(1)
10:                 s ← obtener(t.indiceS.campo, Siguiente(SiguienteSignificado(it).reg))            ▷ O(1)
11:             end if
12:             itl ← CrearIt(SiguienteSignificado(it))                                              ▷ O(1)
13:             EliminarSiguiente(siguiente(itl).reg)                                                ▷ O(1)
14:             if (t.indicesUsados.String) then                                                     ▷ O(1)
15:                 EliminarSiguiente(Siguiente(itl).compadre)                                       ▷ O(1)
16:                 if (Pertenece?(claves(t), indiceS.campo) ∨ longitud(Obtener(indiceS.regpordato, s)) = 1) then   ▷ O(1)
17:                     Borrar(indiceS.regpordato, s)                                                ▷ O(L)
18:                 end if
19:             end if
20:             Borrar(indiceN.regpordato, d)                                                        ▷ O(log(n))
21:             if (mt.indiceN.maximo = d) then                                                      ▷ O(1)
22:                 mt.indiceN.maximo ← Maximo(mt.indiceN.regpordato)                                ▷ O(1)
23:             end if
24:             if (mt.indiceN.minimo = d) then                                                      ▷ O(1)
25:                 mt.indiceN.minimo ← Minimo(mt.indiceN.regpordato)                                ▷ O(1)
26:             end if
27:         end if
28:     else
29:         if (t.indicesUsados.String ∧ t.indiceS.campo = c) then                                   ▷ O(1)
30:             it ← Buscar(t.indiceS.regpordato, d)                                                 ▷ O(L)
31:             if (SiguienteClave(it) = d) then                                                     ▷ O(L)
32:                 if (t.indicesUsados.Nat) then                                                    ▷ O(1)
33:                     n ← obtener(t.indiceN.campo, Siguiente(SiguienteSignificado(it).reg))        ▷ O(1)
34:                 end if
35:                 itl ← CrearIt(SiguienteSignificado(it))                                          ▷ O(1)
36:                 EliminarSiguiente(siguiente(itl).reg)                                            ▷ O(1)
37:                 if (t.indicesUsados.Nat) then                                                    ▷ O(1)
38:                     EliminarSiguiente(Siguiente(itl).compadre)                                   ▷ O(1)
39:                     if (Pertenece?(claves(t), indiceN.campo) ∨ longitud(Obtener(indiceN.regpordato, n)) = 1) then   ▷ O(1)
40:                         Borrar(mt.indiceN.regpordato, n)                                         ▷ O(log(n))
41:                     end if
42:                 end if
43:                 Borrar(indiceS.regpordato, d)                                                    ▷ O(L)
44:                 if (mt.indiceS.maximo = d) then                                                  ▷ O(L)
45:                     mt.indiceS.maximo ← Maximo(mt.indiceS.regpordato)                            ▷ O(1)
46:                 end if
47:                 if (mt.indiceS.minimo = d) then                                                  ▷ O(L)
48:                     mt.indiceS.minimo ← Minimo(mt.indiceS.regpordato)                            ▷ O(1)
49:                 end if
50:             end if
51:         else
52:             BuscaryBorrar (crit, mt)                                                             ▷ O(n*L)
53:         end if
54:     end if
55: end if
```

Complejidad: $O(L + log(n))$ para un campo clave indexado, $O(n * L)$ sino.

Justificación: Como *crit* solo tiene un campo, las funciones Pertenece? y CoincidenTodos son $O(1)$. Este algoritmo cubre 3 casos a grandes rasgos. 1 - (6-27) Si el campo del registro crit que se pasa por parametro esta indexado y es de tipo nat entonces hay que borrar el registro accediendo a regpordato en indiceN, pedir su significado que es una lista de un elemento, ya que, el campo es clave y luego en el elemento de esa lista, acceder al registro por reg y eliminarlo con la funcion del iterador de lista. Si ademas, la tabla estaba indexada por un campo string entonces hay que borrar el registro en la otra estructura mediante la componente compadre. 2- (29-51) Si el campo del registro crit esta indexado y es de tipo string hacer lo mismo que en 1 pero en indiceS. 3- Si el campo del registro crit no esta indexado hay que borrarlo de la lista de registros. Sin embargo , si la tabla estaba indexada tambien hay que borrarlos de las estructuras indiceN e indiceS , sino solo hay que eliminarlo de mt.registros. De esto se encarga la funcion BuscaryBorrar (linea 52).

---

iNombre(**in** mt: `megatab`) → res: nombre

```
1:  res ← mt.nombre
```

Complejidad: O(1)

Justificación: Devuelvo un elemento de la estructura, como es una tubla la complejidad de acceder a el es O(1).

iClaves(**in** mt : `megatab`) $\rightarrow$ res: conj(campo)

1: $res \leftarrow mt.claves$

Complejidad: $O(1)$)
Justificación: Devuelvo un elemento de la estructura, como es una tubla la complejidad de acceder a el es O(1)

BuscarYBorrar(**in** *crit*: `registro`, **in/out** *mt*: `megatab`)

```
1: rs ← crearIt(mt.registros)                                                          ▷ O(1)
2: while (HaySiguiente(rs) && ¬(Obtener(Siguiente(rs), c)) = d)) do                    ▷ O(n*L)
3:     Avanzar(rs)                                                                     ▷ O(1)
4: end while
5: if (mt.indiceUsados.Nat || mt.indicesusados.String) then                            ▷ O(1)
6:     if mt.indiceUsados.Nat then                                                     ▷ O(1)
7:         ls ← Obtener(Obtener(Siguiente(rs), mt.indiceN.campo), mt.indiceN.regpordato)   ▷ O(1)
8:         fa ← CrearIt(ls)                                                            ▷ O(1)
9:         while (HaySiguiente(fa) && Obtener(Siguiente(Siguiente(fa).reg), c) = d) do  ▷ O(1)
10:            Avanzar(fa)                                                             ▷ O(1)
11:        end while
12:        m ← Significado(Siguiente(Siguiente(fa).reg), mt.indiceN.campo)             ▷ O(1)
13:        EliminarSiguiente(fa.reg)                                                   ▷ O(1)
14:        if (longitud(Obtener(Obtener(Siguiente(rs), mt.indiceN.campo)) = 0) then    ▷ O(1)
15:            Borrar(mt.IndiceN.regpordato, Obtener(Siguiente(Siguiente(fa).reg), mt.indiceN.campo))   ▷
   O(log(n))
16:        end if
17:        if (mt.indiceN.maximo = m) then                                             ▷ O(1)
18:            mt.indiceN.maximo ← Maximo(mt.indiceN.regpordato)                       ▷ O(1)
19:        end if
20:        if (mt.indiceN.minimo = m) then                                             ▷ O(1)
21:            mt.indiceN.minimo ← Minimo(mt.indiceN.regpordato)                       ▷ O(1)
22:        end if
23:    end if
24:    if mt.indicesusados.String then                                                 ▷ O(1)
25:        ts ← Obtener(Obtener(Siguiente(rs), mt.indiceS.campo), mt.indiceS.regpordato)   ▷ O(1)
26:        fu ← CrearIt(ts)                                                            ▷ O(1)
27:        while (HaySiguiente(fu) && Obtener(Siguiente(Siguiente(fu).reg), c) = d) do ▷ O(L)
28:            Avanzar(fu)                                                             ▷ O(1)
29:        end while
30:        n ← Significado(Siguiente(Siguiente(fa).reg), mt.indiceS.campo)             ▷ O(1)
31:        EliminarSiguiente(fu.reg)                                                   ▷ O(1)
32:        if (longitud(Obtener(Obtener(Siguiente(rs), mt.indiceS.campo)) = 0) then    ▷ O(1)
33:            Borrar(mt.IndiceS.regpordato, Obtener(Siguiente(Siguiente(fu).reg), mt.indiceS.campo))   ▷ O(L)
34:        end if
35:        if (mt.indiceS.maximo = n) then                                             ▷ O(1)
36:            mt.indiceS.maximo ← Maximo(mt.indiceS.regpordato)                       ▷ O(1)
37:        end if
38:        if (mt.indiceS.minimo = n) then
39:            mt.indiceS.minimo ← Minimo(mt.indiceS.regpordato)                       ▷ O(1)
40:        end if
41:    end if
42: else
43:    EliminarSiguiente(rs)                                                           ▷ O(1)
44: end if
```

Complejidad: O(n*L)

Justificación: Este algoritmo lo utiliza BorrarRegistro cuando el campo del registro criterio no esta indexado . Si la tabla estaba indexada igual hay que borrar el registro en indiceN en el caso de que haya un indice en un campo nat o en indiceS para un indice en campo string. Si no estaba indexada la tabla simplemente busca el registro en la lista de todos los registros que contiene la tabla y cuando lo encuentra lo borra con la operacion del iterador de lista. Este es el peor caso que es O(n*L) donde n es la cantidad de registros y L es la longitud maxima de un dato string.

iIndices(**in** *mt* : `megatab`) → *res*: conj(campo)

```
1: conj(campo) res ← Vacio()                                          ▷ O(1)
2: if mt.indicesUsados.Nat = true then                                ▷ O(1)
3:     AgregarRapido(res, mt.indiceN.campo) O(copy(campo))
4: end if
5: if mt.indicesUsados.String = true then                             ▷ O(1)
6:     AgregarRapido(res, mt.indiceS.campo)                           ▷ O(copy(campo))
7: end if
```

Complejidad: $O(1)$)
Justificación: Como la complejidad de copiar un campo es constante porque este es acotado, luego la complejidad de agregarRapido pasa a ser constante luego la complejidad total pasa a ser O(1)

iCampos(**in** *mt* : `megatab`) → *res*: conj(campo)

```
1: res ← Campos(mt.columnas)
```

Complejidad: $O(1)$)
Justificación: Devuelvo un elemento de la estructura, como es una tubla la complejidad de acceder a el es O(1) y como los campos de un registro están acotados, la función Campos de registros toma O(1)

iRegistros(**in** *mt* : `megatab`) → *res*: conj(campo)

```
1: res ← mt.registros
```

Complejidad: $O(1)$
Justificación: Devuelvo un elemento de la estructura, como es una tubla la complejidad de acceder a el es O(1)

iCantidadDeAccesos(**in** *mt* : `megatab`) → *res*: Nat

```
1: res ← mt.modificaciones
```

Complejidad: $O(1)$)
Justificación: Devuelvo un elemento de la estructura, como es una tubla la complejidad de acceder a el es O(1)

iTipoCampo(**in** *c* : `campo`, **in** *mt* : `megatab`) → *res*: tipo

```
1: ItDicc(campo, dato) it ← Crearit(t.columnas)                       ▷ O(1)
2: while ¬(SiguienteClave(it) = c) do                                 ▷ O(campos(mt.columnas))
3:     Avanzar(it)                                                    ▷ O(1)
4: end while
5: res ← tipo?(SiguienteSignificado(it))                              ▷ O(1)
```

Complejidad: $O(1)$)
Justificación: La funcion, en peor caso, recorre todos los campos del registro pero como estos son acotados, recorrerlos y compararlos pasa a ser constante por lo que la complejidad es O(1)

iIndexar(**in** c: `campo`, **in/out** mt: `megatab`)

```
 1: ItLista(Dicc(campo, dato) it ← CrearIt(mt.registros)                                   ▷ O(1)
 2: int min ← Significado(it, c)                                                           ▷ O(1)
 3: int max ← Significado(it, c)                                                           ▷ O(1)
 4: if TipoCampo(c, mt) then                                                               ▷ O(1)
 5:     mt.indiceUsados.Nat ← true                                                         ▷ O(1)
 6:     mt.indiceN.campo ← c                                                               ▷ O(1)
 7:     while HaySiguiente(it) do                                                          ▷ O(n^2)
 8:         if ¬(MenorOIgual(min, Significado(it, c))) then                                ▷ O(1)
 9:             min ← Significado(it, c)                                                   ▷ O(1)
10:         end if
11:         if (MenorOIgual(max, Significado(it, c))) then                                 ▷ O(1)
12:             max ← Significado(it, c)                                                   ▷ O(1)
13:         end if
14:         if Definido?(mt.indiceN.regpordato, Significado(it, c)) then                   ▷ O(log(n))
15:             AgregarAtras(Obtener(mt.indiceN.regpordato, Significado(it, c)), < it, CrearIt(Vacia()) >)   ▷ O(log(n))
16:         else
17:             Agregar(mt.indiceN.regpordato, Significado(it, c), < it, CrearIt(Vacia()) >)   ▷ O(1)
18:         end if
19:         if mt.IndicesUsados.String then                                                ▷ O(n + log(n) + L)
20:             itLista(apuntador)In ← Retroceder(CrearItUlt(Obtener(mt.indiceN.regpordato, Significado(it, c))))   ▷ O(log(n))
21:             Is ← CrearIt(Obtener(mt.indiceS.regpordato), Significado(it, c))           ▷ O(L)
22:             while Obtener(Siguiente(Siguiente(Is).reg), indiceS.campo) ! = Significado(it, c) do   ▷ O(n)
23:                 Avanzar(Is)                                                            ▷ O(1)
24:             end while
25:             Siguiente(Is).compadre = In                                                ▷ O(1)
26:             Siguiente(In).compadre = Is                                                ▷ O(1)
27:         end if
28:         Avanzar(it)                                                                    ▷ O(1)
29:     end while
30:     mt.indiceN.minimo ← min                                                            ▷ O(1)
31:     mt.indiceN.maximo ← max                                                            ▷ O(1)
32: else
33:     mt.indiceUsados.String ← true                                                      ▷ O(1)
34:     mt.indiceS.campo ← c                                                               ▷ O(1)
35:     while HaySiguiente(it) do                                                          ▷ O(n^2)
36:         if ¬(MenorOIgual(min, Significado(it, c))) then                                ▷ O(1)
37:             min ← Significado(it,c)                                                    ▷ O(1)
38:         end if
39:         if (MenorOIgual(max, Significado(it,c ))) then                                 ▷ O(1)
40:             max ← Significado(it, c)                                                   ▷ O(1)
41:         end if
42:         if Definido?(mt.indiceS.regpordato, Significado(it, c) then                    ▷ O(L)
43:             AgregarAtras(Obtener(mt.indiceS.regpordato, Significado(it, c)), < it, CrearIt(Vacia()) >)   ▷ O(L)
44:         else
45:             Agregar(mt.indiceS.regpordato, Significado(it, c), < it, CrearIt(Vacia()) >)   ▷ O(1)
46:         end if
47:         if mt.IndicesUsados.Nat then                                                   ▷ O(n + log(n) + L)
48:             itLista(apuntador)Is ← Retroceder(CrearItUlt(Obtener(mt.indiceS.regpordato, Significado(it, c))))   ▷ O(L)
49:             In ← CrearIt(Obtener(mt.indiceN.regpordato), Significado(it, c))           ▷ O(log(n))
50:             while Obtener(Siguiente(Siguiente(In).reg), indiceN.campo) ! = Significado(it, c) do   ▷ O(n)
51:                 Avanzar(In)                                                            ▷ O(1)
52:             end while
53:             Siguiente(In).compadre = Is                                                ▷ O(1)
54:             Siguiente(Is).compadre = In                                                ▷ O(1)
55:         end if
56:         Avanzar(it)                                                                    ▷ O(1)
57:     end while
58:     mt.indiceS.minimo ← min                                                            ▷ O(1)
59:     mt.indiceS.maximo ← max                                                            ▷ O(1)
60: end if
```

Complejidad: $O(n^2)$

Justificación: Si el índice a indexar es Nat, entra al primer if (linea 4) y en el peor caso, ya hay índice String y tengo que hacer que el compadre de todo registro apunte a su paralelo en el otro índice, y eso en el peor caso sería O(n^2), lo mismo para el caso en el cual el índice a indexar es String.

iMismosTipos(**in** r : `registro`, **in** mt : `megatab`) → res: bool

1: ItDicc(campo, dato) it_1 ← CrearIt(r) ▷ $O(1)$
2: **while** (HaySiguiente(it)) && (TipoCampo(SiguienteClave(it), t) = Nat?(SiguienteSignificado(it))) **do** ▷ $O(\#(campos(r))$
3: Avanzar(it) ▷ $O(1)$
4: **end while**
5: res ← ¬ (HaySiguiente(it)) ▷ $O(1)$

Complejidad: $O(1)$)
Justificación: Como la cantidad de campos de un registro es acotada, y estos a su vez son acotados la complejidad de recorrelos es constante, luego la complejidad es O(1)

iCompatible(**in** r : `registro`, **in** mt : `megatab`) → res:bool

1: res ← ((campos(r) = campos(mt.columnas)) && MismosTipos(r,t)) ▷ $O(\#(campos(r)))$

Complejidad: $O(1)$
Justificación: Como los campos son acotados y estos mismos tambien lo son la complejidad de la comparacion es constante, ademas la complejidad de MismosTipos(r,t) es constate, con lo que la complejidad termina siendo O(1)

iDameColumna(**in** c : `campo`, **in** cr : `conj(registro)` → res: conj(dato)

1: res ← Vacio() ▷ $O(1)$
2: Conj(Dicc(campo, clave)) it ← Crearit(cr) ▷ $O(1)$
3: **while** HaySiguiente(it) **do** ▷ $O(\#(cr)^2)$
4: Agregar(res, Significado(c, Siguiente(it)) ▷ $O()$
5: Avanzar(it)
6: **end while**

Complejidad: $O(n^2)$
Justificación: Recorre todos los registros de cr y usa agregar de conj de los modulos basicos cuya complejidad es la suma de las comparaciones, como estoy comparando tipo string tomo a L como el mas largo, luego la suma es constante porque los campos son acotados. Entonces la complejidad pasa a ser $O(n^2 * L)$ porque en el peor de los casos recorro dos veces el mismo conjunto

iMaximo(**in** c : `campo`,**in** mt : `megatab`) → res: dato

1: **if** TipoCampo(c, mt) **then**
2: res ← mt.indiceN.maximo
3: **else**
4: res ← mt.indiceS.maximo
5: **end if**

Complejidad: $O(1)$)
Justificación:Devuelvo un elemento de la estructura, como es una tubla la complejidad de acceder a el es O(1) y ademas pregunta el tipo de un campo cuya complejidad es O(1)

iMinimo(**in** c: `campo`,**in** mt: `megatab`) → res: dato

```
1: if TipoCampo(c, mt) then
2:     res ← mt.indiceN.minimo
3: else
4:     res ← mt.indiceS.minimo
5: end if
```

Complejidad: $O(1)$)
Justificación:Devuelvo un elemento de la estructura, como es una tubla la complejidad de acceder a el es O(1) y ademas pregunta el tipo de un campo cuya complejidad es O(1)

iBuscarT(**in** t: `megatab`, **in** $crit$: `registro`) $\rightarrow$ res : conj(registro)

1: $itDicc(campo, dato)$ $it \leftarrow CrearIt(crit)$
2: **while** $\big(HaySiguiente(it) \wedge Def?(SiguienteClave(it), t.columnas) \wedge Nat?(SiguienteSignificado(it)) = tipoCampo(SigienteClave(it), t)\big)$ **do** ▷ $O(min(\#Campos(crit), \#Campos(t.columnas))) = O(1)$ pues la cantidad de campos de la tabla está acotada
3: $Avanzar(it)$ ▷ $O(1)$
4: **end while**
5: $res \leftarrow Vacio()$ ▷ $O(1)$
6: $conj(campo)$ $cs \leftarrow campos(crit)$ ▷ Pasado por referencia $O(1)$
7: **if** $\neg HaySiguiente(it)$ **then** ▷ Si hay siguiente, algun campo no coincidió y la busqueda no tiene sentido
8: **if** $(t.indicesUsados.Nat = true \wedge Pertenece?(t.indiceN.campo, cs))$ **then**
9: $ItLog(nat, Lista(apuntadores))$ $i \leftarrow Buscar(t.indiceN.regpordato, n)$ ▷ $O(log(n))$
10: $ItLista(apuntadores)$ $I \leftarrow CrearIt(SiguienteSignificado(i))$ ▷ $O(1)$
11: **while** $HaySiguiente(I)$ **do** ▷ $O(n)$
12: **if** $CoindicenTodos(Siguiente(Siguiente(I).reg), cs, crit)$ **then** ▷ $O(1)$
13: $AgregarRapido(res, Siguiente(Siguiente(I).reg))$ ▷ $O(1)$
14: **end if**
15: $Avanzar(I)$ ▷ $O(1)$
16: **end while**
17: **else**
18: **if** $(t.indicesUsados.String = true \wedge Pertenece?(t.indiceS.campo, cs))$ **then** ▷ $O(L)$
19: $ItStr(conj(itLista(registro)))$ $i \leftarrow Buscar(t.indiceS.regpordato,)$ ▷ $O(L)$
20: $ItLista(itLista(registro))$ $I \leftarrow CrearIt(SiguienteSignificado(i))$ ▷ $O(1)$
21: **while** $HaySiguiente(I)$ **do** ▷ $O(n * L)$
22: **if** $CoindicenTodos(Siguiente(Siguiente(I).reg), cs, crit)$ **then** ▷ A lo sumo comparo el string más largo // $O(L)$
23: $AgregarRapido(res, Siguiente(Siguiente(I).reg))$ ▷ $O(1)$
24: **end if**
25: $Avanzar(I)$ ▷ $O(1)$
26: **end while**
27: **else**
28: $ItLista(registro)$ $I \leftarrow CrearIt(t.registros)$ ▷ $O(1)$
29: **while** $HaySiguiente(I)$ **do** ▷ Recorro todos los registros // $O(n * L)$
30: **if** $CoindicenTodos(Siguiente(I), cs, crit)$ **then** ▷ A lo sumo comparo el string más largo // $O(L)$
31: $AgregarRapido(res, Siguiente(I))$ ▷ $O(1)$
32: **end if**
33: $Avanzar(I)$ ▷ $O(1)$
34: **end while**
35: **end if**
36: **end if**
37: **end if**

Complejidad: $O(L + log(n))$ para un campo clave indexado, $O(n * L)$ sino
Justificación: Asumo que la cantidad de campos en *crit* está acotada similarmente a la cantidad de campos en la tabla t, pues sino habría inevitablemente una complejidad asociada a la longitud de *crit*. Por lo tanto, la longitud de *cs* también está acotada y las funciones Pertenece? y CoincidenTodos son $O(1)$. En el caso de que *crit* contenga un campo clave indexado, los ciclos de las líneas 11-16 y 21-26 solo ocurren una vez, dado que el conjunto de registros asociados al dato es unitario, por lo que su complejidad es $O(1)$ y $O(L)$, respectivamente. Entonces, el *then* de la linea 8-17 tiene complejidad $O(log(n))$ y el *else* de 18-36 es $O(L)$, resultando la complejidad del *if* de las lineas 7-37 la suma de los dos $O(log(n) + L)$

iEsta(**in** r : `registro`, **in** mt : `megatab`) → res: bool

```
1:  if (¬compatible(r, t)) then                                                    ▷ O(1)
2:      res ← false                                                                ▷ O(1)
3:  else
4:      if (mt.indicesUsados.Nat = true || mt.indicesUsados.String = true) then    ▷ O(1)
5:          if ((mt.indiceUsados.Nat) & & Pertenece?(mt.claves,mt.indiceN.campo)) then   ▷ O(1)
6:              if (Definido?(mt.indiceN.regpordato,obtener(mt.indiceN.campo,r)) then    ▷ O(log(n))
7:                  res ← r = primero(siguiente(significado(mt.indiceN.regpordato))).reg ▷ O(L)
8:              end if
9:          end if
10:         if ((mt.indiceUsados.String) & & (Pertenece?(mt.claves,mt.indiceN.campo))) then  ▷ O(1)
11:             if Definido?(mt.indiceS.regpordato,obtener(mt.indiceS.campo,r)) then     ▷ O(L)
12:                 res ← r = primero(siguiente(significado(mt.indiceS.regpordato))).reg ▷ O(L)
13:             end if
14:         end if
15:     else
16:         i ← crearIt(mt.registros)                                              ▷ O(1)
17:         res ← false                                                            ▷ O(1)
18:         while (HaySiguiente(i)) do                                             ▷ O(1*n*L)
19:             res ← (res || siguiente(i) = r)                                    ▷ O(1)
20:             Avanzar(i)                                                         ▷ O(1)
21:         end while
22:     end if
23: end if
```

Complejidad: $\Theta(log(n) + L)$
Justificación: res es true sii el registro esta en la tabla. Si el registro tienen algun campo que no esta en tabla es false. Si la tabla esta indexada por un campo nat y además el campo es clave entonces pregunta si el dato en ese campo coincide con el del registro en el mismo lugar. Luego pregunta si el resto del registro coincide usando el significado del de regpordato. Buscar en un diccLog cuesta O(log(n)) Mismo caso para un string donde buscar un registro cuesta O(L).En el caso donde no hay indices en la tabla busca en los registros en un conjunto lineal que en peor caso es O(n*L)

---

iCombinarRegistros(**in** c : `campo`,**in** t_1 : `megatab`, **in** t_2 : `megatab`) → res: conj(registro)

```
1:  res ← Vacio()                                  ▷ Vacio del módulo conjunto lineal // O(1)
2:  ItConj(Dicc(campo, dato)) it ← CrearIt(Registros(t1))   ▷ O(1)
3:  while HaySiguiente(it) do
4:      dato valor ← Obtener(HaySiguiente(it), c)   ▷ O(1)
5:      registro crit ← Definir(c, d, Vacio())     ▷ O(1)
6:      ItConj(registros) Ic ← CrearIt(BuscarT(t2, crit))
7:      while HaySiguiente(Ic) do
8:          Agregar(res, AgregarCampos(Siguiente(it), Siguiente(IC)))   ▷ O(L)
9:      end while
10: end while
```

Complejidad: Para un campo c indice de t_1 y t_2 O((n+m)*(L+log(m+n))
Justificación: Ambos terminos $n + m$ representan en realidad $max(n, m)$. El ciclo externo de las líneas 3-9 itera siempre n veces (con n la cantidad de registros en t_1). Sin embargo, en peor caso la complejidad del BuscarT de la línea 6 es lineal en m, al igual que el ciclo de las líneas 7-9. Por lo tanto, en peor caso es O(n*m*L) por el costo de comparación de la linea 8.

4 Base de Datos

4.1 Especificación

Se utiliza el *TAD BASEDEDATOS* especificado por la práctica.

4.2 Interfaz

Género base
se explica con: Registro , Campo , Dato, Tabla , Conjunto (σ) , Secuencia(σ)

Tablas(**in** b: `base`) $\rightarrow$ *res* : `conj(string)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ tablas(db)}
Complejidad: O(T)
Descripción: Devuelve en un conjunto todos los nombres de las tablas

dameTabla(**in** t: `string`, **in** b: `base`) $\rightarrow$ *res* : `tabla`
Pre $\equiv$ { t $\in$ tablas(b)}
Post $\equiv$ {res $=_{\text{obs}}$ damTabla(t,b)}
Complejidad: O(1)
Descripción: Devuelve la tabla en la base b con el nombre t

HayJoin?(**in** t_1 : `string`, **in** t_2 : `string`, **in** b: `base`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {$t_1 \neq t_2 \wedge \{t_1, t_2\} \subseteq tablas(b)$}
Post $\equiv$ {res $=_{\text{obs}}$ hayJoin?(t_1,t_2,b)}
Complejidad: O(1)
Descripción: res es true sii hay un Join entre la tabla t_1 y t_2

campoJoin(**in** t_1 : `string`, **in** t_2 : `string`, **in** b: `base`) $\rightarrow$ *res* : `campo`
Pre $\equiv$ {hayJoin?(t_1,t_2,b)}
Post $\equiv$ {res $=_{\text{obs}}$ campoJoin(t_1,t_2,b)}
Complejidad: O(1)
Descripción: devuelve el campo por el cual t_1 y t_2 esta joineados

NuevaBD() $\rightarrow$ *res* : `base`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ nuevaBD()}
Complejidad: O(1)
Descripción: Crea una base de datos

AgregarTabla(**in** t: `tabla`, **in/out** b: `base`)
Pre $\equiv$ {$\varnothing$?(registros(t) $\wedge$ b $=_{\text{obs}}$ b_o}
Post $\equiv$ {b $=_{\text{obs}}$ agregarTabla(t, b_o)}
Complejidad: O(1)
Descripción: Agrega una tabla vacía a la base de datos

InsertarEntrada(**in** r: `registro`, **in** t: `string`, **in/out** b: `base`)
Pre $\equiv$ {t $\in$ tablas(b) $\wedge_{\text{L}}$ puedoInsertar?(r, t) $\wedge$ b $=_{\text{obs}}$ b_o}
Post $\equiv$ {b $=_{\text{obs}}$ insertarEntrada(r, t, b_o)}
Complejidad: O(T*L+in)
Descripción: Inserta un registro en una tabla de la base de datos

Borrar(**in** r: `registro`, **in** t: `string`, **in/out** b: `base`)
Pre $\equiv$ {$\sharp$(campos(r)) = 1 $\wedge_{\text{L}}$ dameUno(campos(r)) $\in$ claves(t) $\wedge$ t $\in$ tablas(b) $\wedge$ b $=_{\text{obs}}$ b_o}
Post $\equiv$ {b $=_{\text{obs}}$ borrar(r, t, b_o)}
Complejidad: O(T*L+in)
Descripción: Borra un registro de la tabla según el criterio

GenerarVistaJoin(**in** t_1 : `string`, **in** t_2 : `string`, **in** c: `campo`, **in/out** b: `base`)
Pre $\equiv$ {$\neg(t_1 =_{obs} t_2) \wedge \{t_1, t_2\} \subseteq$ tablas(b) $\wedge_L$ c $\in$ claves(dameTabla(t_1, b)) $\wedge$ c $\in$ claves(dameTabla(t_2, b)) $\wedge$ $\neg$hayJoin?(t_1, t_2, b) $\wedge$ b $=_{obs}$ b_o}
Post $\equiv$ {b $=_{obs}$ generarVistaJoin(t_1, t_2, c, b_o)}
Complejidad: O((n+m)*(L+log(n+m))
Descripción: Genera el join entre 2 tablas de la base de datos

BorrarJoin(**in** t_1 : `tabla`, **in** t_2 : `tabla`, **in/out** b: `base`)
Pre $\equiv$ {hayJoin?(t_1, t_2) $\wedge$ b $=_{obs}$ b_o}
Post $\equiv$ {b $=_{obs}$ borrarJoin(t_1, t_2, b_o)}
Descripción: Borra el join entre 2 tablas de la base de datos

Buscar(**in** *criterio*: `registro`, **in** t: `tabla`, **in** b: `base`) $\rightarrow$ *res* : `conj(registro)`
Pre $\equiv$ {t $\in$ tablas(b)}
Post $\equiv$ {res $=_{obs}$ buscar(criterio, t, b)}
Complejidad: O(L+log(n)) en caso promedio, si hay indice en un campo Nat
Descripción: Devuelve los registros filtrados por criterio

VistaJoin(**in** t_1 : `string`, **in** t_2 : `string`, **in** b: `base`) $\rightarrow$ *res* : `conj(registro)`
Pre $\equiv$ {hayJoin?(t_1, t_2, b)}
Post $\equiv$ {res $=_{obs}$ vistaJoin(t_1, t_2,b)}
Complejidad: R*(L+log(n*m))
Descripción: Visualiza el join entre las tablas

Registros(**in** t: `string`, **in** b: `base`) $\rightarrow$ *res* : `conj(registro)`
Pre $\equiv$ {$t \in tablas(b)$}
Post $\equiv$ {res $=_{obs}$ registros(t,b)}
Complejidad: O(1)
Descripción: Devuelve los registros de una tabla

CantidaDeAccesos(**in** t: `string`, **in** b: `base`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {$t \in tablas(b)$}
Post $\equiv$ {res $=_{obs}$ cantidadDeAccesos(t,b)}
Complejidad: O(1)
Descripción: Devuelve los registros de una tabla

4.3 Pautas de implementación

Representación

4.3.1 Justificación

La Base de datos guarda las tablas en *tablas*, que es un conjunto lineal. *TporNombre* es un diccString donde las claves son los nombres de las tablas y sus significados son iteradores a *tablas*. Se guardan los nombres de las tablas en un conjunto lineal *nombres*. *TablaMax* guarda un iterador de la tabla con mayor cantidad de accesos en *tablas*. *Joins* representa un diccString donde las claves son los nombres de las tablas que tiene joins. Los significados son un diccString donde las claves son los nombres de las tablas que estan joineadas a la tabla anterior(la clave del significado). Su signficado es un *Join*, que tiene el campo por el cual estan joineados. *Cambios* es una lista donde en cada elemento guarda los registros modificados y en que tabla se modificaron. Por ultimo, *verJoin* muestra la visualizacion del Join

entre las tablas con un conjunto lineal de registro.

`BaseDeDatos` **se representa con** `archibase`

donde `archibase` es `tupla`(*tablas*: `conj(tabla)` ,
TporNombre: `diccString(string, itConj(tabla))` ,
nombres: `conj(string)` ,
tablaMax: `itConj(tabla)` ,
Joins: `diccString(string, diccString(string, Join))` ,
LosJoins: `lista(tabla))`)

donde `Join` es `tupla`(*campo*: `str` ,
cambios: `lista(tupla<tabmod: string,regmod: registro>)` ,
verJoin: `itLista(tabla)`)

4.3.2 Invariante de representación

Informal

(1) Una tabla está en *e.tablas* sí y sólo sí su nombre está en *e.nombres*
(2) Todo significado de *e.TporNombre* apunta a una tabla cuyo nombre es la clave
(3) Las claves de *e.TporNombre* son *e.nombres*
(4) *e.tablaMax* apunta a la tabla de *e.tablas* con más modificaciones
(5) Las claves de *e.Joins* y las claves de los significados de *e.Joins* están en *e.nombres*
(6) Dentro del significado del significado de *e.Joins* el campo es clave de las claves internas y externas de *e.Joins*
(7) Los cambios del significado del significado de *e.Joins* guardan en una lista de tuplas, donde tabmod es la tabla en la que se modifico los registros y regmod es el registro agregado o eliminado.
(8) El verjoin del significado del significado de *e.Joins* apunta a la tabla en *e.LosJoins* visualizados o generados por última vez entre las claves

Formal

Rep : archibase a $\longrightarrow$ bool
Rep(a) $\equiv$ true $\Longleftrightarrow$

(1) $(\forall t : tabla)t \in a.tabla \Rightarrow_{\text{L}} nombre(t) \in a.nombres \wedge (\forall n : string)n \in a.nombres \Rightarrow_{\text{L}} dameTabla(n, a) \in a.tablas \wedge_{\text{L}}$
(2) $(\forall s : string)s \in claves(a.TporNombre) \Rightarrow_{\text{L}} siguiente(obtener(a.TporNombre, s)) \in a.tablas \wedge_{\text{L}}$
(3) $claves(a.TporNombre \subseteq a.nombres \wedge a.nombres \subseteq claves(a.TporNombre) \wedge_{\text{L}}$
(4) $siguiente(a.tablaMax) \wedge_{\text{L}} esMaxima(siguiente(a.tablaMax), a.tablas) \wedge_{\text{L}}$
(5) $claves(a.joins) \subseteq a.nombres \wedge_{\text{L}} (\forall s : string)s \in claves(a.joins) \Rightarrow_{\text{L}} claves(obtener(a.TporNombre, s)) \subseteq a.nombres \wedge_{\text{L}}$
(6) $(\forall s : string)s \in claves(a.joins) \Rightarrow_{\text{L}} (\forall s' : string)s' \in claves(obtener(a.TporNombre, s)) \Rightarrow_{\text{L}} (\forall t : tabla)nombre(t) = s \Rightarrow_{\text{L}} campo(obtener(obtener(a.TporNombre, s), s')) \in claves(t) \wedge$
$(\forall t' : tabla)nombre(t') = s' \Rightarrow_{\text{L}} campo(obtener(obtener(a.TporNombre, s), s')) \in claves(t') \wedge_{\text{L}}$
(8)$(\forall s : string)s \in claves(a.joins) \Rightarrow_{\text{L}} (\forall s' : string)s' \in claves(obtener(a.TporNombre, s)) \Rightarrow_{\text{L}}$
$siguiente(VerJoin(obtener(obtener(a.TporNombre, s), s'))) \in a.LosJoins$

4.3.3 Predicado de abstracción

4.4 Algoritmos

Algoritmos

Trabajo Práctico 2Algoritmos del módulo
En todos los algoritmos se nota que : n y m es la cantidad de registros en la tabla. L es la máxima longitud de un valor string de un registro en la tabla pasada por parámetro. T es la cantidad de tablas en la base de datos.

itablas(**in** a : `archibase`) $\rightarrow$ res : conj(string)

1: $res \leftarrow Vacio()$ ▷ O(1)
2: $i \leftarrow creatIt(a.TporNombre)$ ▷ O(1)
3: **while** HaySiguiente(i) **do** ▷ O(1)
4: $Agregar(res, SiguienteClave(i))$ ▷ O(1)
5: **end while**

Complejidad: O(T)
Justificación: El ciclo recorre los nombres de la tabla en a.TporNombre y los va agregando a un conjunto lineal con su operacion del modulo. Si T es la cantidad de tablas, entonces el ciclo siempre itera T veces.

---

idameTabla(**in** t_1 : `string` **in** a : `archibase`) $\rightarrow$ res : megatab

1: $res \leftarrow Siguiente(Obtener(a.TporNombre, t_1))$ ▷ O(1)

Complejidad: O(1)
Justificación: Busca la tabla en el diccString y devuelve un iterador (de conjunto lineal) a su significado que es la tabla y pido su siguiente para devolverla. Como el nombre de las tablas está acotado, la complejidad de Obtener es O(1)

---

iNuevaBD() $\rightarrow$ res : archibase

1: $res.tabla \leftarrow Vacio()$ ▷ Vacio() del módulo Conjunto lineal // $O(1)$
2: $res.TporNombre \leftarrow Vacio()$ ▷ Vacio() del módulo DiccString // $O(1)$
3: $res.nombres \leftarrow Vacio()$ ▷ Vacio() del módulo Vector (string) // $O(1)$
4: $res.tablaMax \leftarrow CrearIt(Vacio())$ ▷ Vacio() del módulo Conjunto lineal // $O(1)$
5: $res.joins \leftarrow Vacio()$ ▷ Vacio() del módulo DiccString // $O(1)$
6: $res.Losjoins \leftarrow Vacia()$ ▷ Vacia() del módulo Lista enlazada // $O(1)$

Complejidad: $O(1)$

---

iAgregarTabla(**in** mt : `megatab` , **in/out** a : `archibase`)

1: $AgregarRapido(a.tablas, mt)$ ▷ O(1)
2: $i \leftarrow CrearIt(a.tablas)$
3: **while** $HaySiguiente(i) \wedge Siguiente(i) \neq mt$ **do**
4: $Avanzar(i)$
5: **end while**
6: $Definir(a.TporNombre, nombre(mt), i)$ ▷ O(1)
7: $AgregarRapido(a.nombres, nombre(mt))$ ▷ $\Theta(copy(mt))$
8: **if** $(a.tablas = vacio())$ **then**
9: $a.TablaMax \leftarrow i$ ▷ O(1)
10: **else**
11: **if** $(CantidadAccesos(nombre(mt), a) > CantidadAccesos((nombre(siguiente(a.TablaMax)))), a)$ **then**
12: $a.TablaMax \leftarrow i$ ▷ O(1)
13: **end if**
14: **end if**

Complejidad: $\Theta(copy(mt))$
Justificación: Definir de diccString es O(1) (dado que los nombres de las tablas estan acotadas) y AgregarRapido es de conjunto lineal que cuesta $\Theta(copy(mt))$.

---

iInsertarEntrada(**in** r : `dicc(campo,dato)` , **in** t : `string`, **in/out** a : `archibase`)

```
1: i ← Buscar(a.TporNombre, t)                                                    ▷ O(1)
2: AgregarRegistro(Siguiente(i), r)                                               ▷ O(L+ in)
3: if (CantidadAccesos(Siguiente(Obtener(a.TporNombre, t)), a) > CantidadAccesos(Siguiente(a.TablaMax), a))
   then                                                                           ▷ O(1)
4:     a.TablaMax ← obtener(a.TporNombre, t)                                      ▷ O(1)
5: end if
6: if Definido?(a.joins, t) then
7:     i ← crearIt(obtener(a.joins, t))                                           ▷ O(1)
8:     while (HaySiguiente(i)) do                                                 ▷ O(T)
9:         AgregarAdelante(SiguienteSignificado(i).cambios, < t, r >)             ▷ O(1)
10:        if (iHayJoin(siguienteClave(i), t, a) then                             ▷ O(1)
11:            AgregarAdelante(Obtener(Obtener(a.joins, SiguienteClave(i)), t).cambios, < t, r >)   ▷ O(1)
12:        end if
13:        Avanzar(i)                                                             ▷ O(1)
14:    end while
15: end if
```

Complejidad: O($T * L + in$)
Justificación: Las lineas 1-2 busca la tabla en TporNombre y luego agrega el registro pasado por parametro con la operacion AgregarRegistro del modulo Tabla. Luego se fija si con esta modificacion supera a la Tabla Maxima de la base de datos que eso lo hace en O(1). Luego las lineas 6-15 se fija si la tabla tiene Joins (O(T) en peor caso) y actualiza los cambios debido a la nueva insercion. Ademas si alguna de las tablas con las que tenia join esa tambien tenia un join con la tabla pasada por parametro , tambien la actualiza en sus cambios del join. Esto me da O(T+L+in) = O(T*L + in) ya que la primera esta contenida en la otra. in es O(log(n)) en promedio , si hay indice sobre un campo de tipo nat, O(1) sino.

---

iBuscar(**in** r : `dicc(campo,dato)` , **in** s : `string`, **in** a : `archibase`)→ res:conj(registro)

```
1: res ← BuscarT(DameTabla(t, a), r)
```

Complejidad: Para un campo clave indexado en r $O(T * L + in)$, $O(n * L)$ sino
Justificación: En el caso de un campo clave indexado en r, la complejidad de BuscarT (del módulo tabla) es $O(L + log(n))$, el cual está contenido (por algebra de complejidades) en $O(T * L + in)$. Sino, BuscarT tiene complejidad $O(n * L)$.

iBorrar(**in** r : `dicc(campo,dato)` , **in** t : `string`, **in/out** a : `archibase`)

1: $i \leftarrow Buscar(a.TporNombre, t)$ ▷ O(1)
2: $BorrarRegistro(siguientesignificado(i), r)$ ▷ O(L+ in)
3: **if** $CantidadAccesos((siguiente(significado(a.TporNombre, t)), a) > CantidadAccesos(siguiente(a.TablaMax), a))$ **then**
4: $a.TablaMax \leftarrow obtener(a.TporNombre, t)$ ▷ O(1)
5: **end if**
6: **if** $(Definido?(a.joins), t)$ **then**
7: $i \leftarrow crearIt(obtener(a.joins, t))$ ▷ O(1)
8: **while** $(HaySiguiente(i))$ **do** ▷ O(T)
9: $AgregarAdelante(siguientesignificado(i).cambios, < t, r >)$ ▷ O(1)
10: **if** $(iHayJoin(siguienteclave(i), t, a)$ **then**
11: $AgregarAdelante(obtener(obtener(a.joins, siguienteclave(i)), t).cambios, < t, r >)$ ▷ O(1)
12: **end if**
13: Avanzar(i)
14: **end while**
15: **end if**

Complejidad: O(T * L + in)
Justificación: Borrar el registro de la tabla toma L+log(n) porque llamamos a la operación BorrarRegistro de tabla, y luego para actualizar los cambios de los joins en el peor caso la tabla t estaría joineada a todas las otras tablas por lo que habría que recorrerlas todas y sería O(T). Esto me da O(T+L+in) = O(T*L + in) ya que la primera esta contenida en la otra. in es O(log(n)) en promedio , si hay indice sobre un campo de tipo nat, O(1) sino.

---

itablaMaxima(**in** a : `architabla`) → res : string

1: $res \leftarrow Siguiente(a.tablaMax).nombre$ ▷ O(1)

Complejidad: O(1)
Justificación: tablaMax devuelve un iterador a la tabla maxima , luego pido siguiente para obtener la tabla y obtengo su nombre accediendo con .nombre

---

iHayJoin?(**in** t_1 : `string`, **in** t_2 : `string` , **in** a : `archibase`) → res :bool

1: $res \leftarrow Definido?(Obtener(a.Joins, t_1), t_2)$ ▷ $O(L) + O(L) = O(L)$

Complejidad: O(1)
Justificación: Como L es el nombre más largo de una tabla y los nombre están acotados, $O(L)$ resulta $O(1)$.

---

iborrarJoin(**in** t_1 : `string`, **in** t_2 : `string` , **in** a : `archibase`)

1: $j \leftarrow Buscar(Obtener(a.Joins, t_1), t_2)$ ▷ O(1)
2: $EliminarSiguiente(j)$ ▷ O(1)

Complejidad: O(1)
Justificación: Creo un iterador usando el Buscar para encontrar el t_2 que es que que esta "joineado" a t_1. Luego borro (usando el la funcion del iterador diccString) t_2 de las tablas "joineadas" con t_1.

---

icampoJoin(**in** t_1 : `string`, **in** t_2 : `string` , **in** a : `archibase`) → res : campo

1: $j \leftarrow Buscar(Obtener(a.Joins, t_1), t_2)$ ▷ O(1)
2: $res \leftarrow SiguienteSignificado(j).campo$ ▷ O(1)

Complejidad: O(1)
Justificación: Creo un iterador usando el Buscar para encontrar el t_2 que es que que esta "joineado" a t_1. Luego accedo a su significado mediante siguientesignificado de diccString y devuelvo la componente campo de la tupla.

---

igenerarVistaJoin(**in** t_1 : `string`, **in** t_2 : `string` , **in** ca : `campo` , **in** a : `archibase`) $\rightarrow$ res : itLista(registro)

1: $rs \leftarrow combinarRegistros(ca, t_1, t_2)$ ▷ O((n+m)*(L+log(n+m))
2: $it \leftarrow crearIt(rs)$
3: $nt \leftarrow NuevaTabla(\text{"}nuevat\text{"}, Agregar(vacio(), campos(it)))$
4: $Indexar(ca, nt)$ ▷ Como no hay registros, es O(1)
5: **while** $haySiguiente(it)$ **do** ▷ O(n+m)*O(l+log(n+m))=O((n+m)*(L+log(n+m)))
6: $AgregarRegistro(Siguiente(it), nt)$ ▷ Hay un indice // O(L+log(n+m))
7: **end while**
8: $AgregarAdelante(nt, a.LosJoins)$ ▷ O(n*L)
9: $it2 \leftarrow crearIt(a.LosJoins)$ ▷ O(1)
10: $res \leftarrow CrearIt(Registros(Siguiente(it2)))$
11: **if** $(\neg Definido?(a.joins, t_1))$ **then**
12: $diccString(join)\ d \leftarrow Definir(Vacio(), t_2, < ca, <>, it >)$ ▷ O(1)
13: $Definir(a.joins, t_1, d)$ ▷ O(1)
14: **else**
15: $Definir(significado(a.joins, t_1), t_2, < ca, <>, it >)$ ▷ O(1)
16: **end if**

Complejidad: O((n+m)*(L+log(n+m)))
Justificación: Primero copio los campos de t_1 para poder usar combinarRegistros del modulo Registro que modifica el primer conjunto de registro pasado por parametro (que serian los registros de t_1). Luego lo agrego adelante con la operacion de lista AgregarAdelante que devuelve un iterador a la lista. La linea 5 en adelante se encarga de agregarlo a los joins en los casos que t_1 no tenia ningun join antes o agregar un join mas en t_1 con t_2.

iVistaJoin(**in** t_1 : `string`, **in** t_2 : `string` **in** a : `archibase`) → res : itLista(registro)

1: $tab \leftarrow Siguiente(Obtener(Obtener(a.Joins, t_1), t_2).verJoin)$ ▷ O(L)
2: $itC \leftarrow CrearItUlt(Obtener(Obtener(a.Joins, t_1), t_2).cambios)$ ▷ O(L)
3: $registro\ crit \leftarrow Vacio()$ ▷ O(1)
4: $string\ ca \leftarrow CampoJoin(t_1, t_2, a)$ ▷ O(1)
5: **while** $hayAnterior(itC)$ **do** ▷ O(R * (L + log(n * m)))
6: $dato\ d \leftarrow Obtener(ca, Anterior(itC).regmod)$ ▷ O(1)
7: $Definir(ca, d, crit)$ ▷ O(L)
8: **if** $Esta(Anterior(itC).regmod, Anterior(itC).tabmod)$ **then** ▷ significa que el registro se agregó O(L + log(n*m)) en promedio
9: **if** $Anterior(itC).tabmod = t_1$ **then** ▷ O(L + log(m) + log(n)) = O(L + log(n * m)) en promedio
10: **if** $\neg EsVacio?(Buscar(crit, t_2, a))$ **then** ▷ O(L + log(m)) + O (L + log(t)) = O(L + log(m)) en promedio
11: $itB2 \leftarrow CrearIt(Buscar(crit, t_2, a))$ ▷ O(1)
12: $registro\ reg1 \leftarrow AgregarCampos(Anterior(itC).regmod, Siguiente(itB2))$ ▷ O(L)
13: $AgregarRegistro(reg1, tab)$ ▷ O(L+log(t)) en promedio
14: **end if**
15: **else**
16: **if** $\neg EsVacio(Buscar(crit, t_1, a))$ **then** ▷ O(L + log(n)) + O(L + log(t)) = O(L + log(n)) en promedio
17: $itB1 \leftarrow CrearIt(Buscar(crit, t_1, a))$ ▷ O(L + log(n))
18: $registro\ reg2 \leftarrow AgregarCampos(Anterior(itC).regmod, Siguiente(itB1))$ ▷ O(L)
19: **end if**
20: $AgregarRegistro(reg2, tab)$ ▷ O(L + log(t)) en promedio
21: **end if**
22: **else** ▷ Significa que el registro se borró O()
23: **if** $\neg EsVacio?(BuscarT(crit, tab))$ **then** ▷ O(L + log(n)) en promedio
24: $Borrar(tab, crit)$ ▷ O(L + log(n)) en promedio
25: **end if**
26: **end if**
27: $Retroceder(itC)$ ▷ O(1)
28: **end while**
29: $res \leftarrow CrearIt(Registros(tab))$

Complejidad: O(R * (L + log(n * m)))
Justificación: Recorremos la lista de cambios en ambas tablas (R), definimos el cambio fue agregado o borrado y ahí vemos

---

iregistros(**in** t_1 : `string` **in** a : `archibase`) → res : conj(registros)

1: $res \leftarrow registros(dameTabla(t_1))$ ▷ O(1)

Complejidad: O(1)
Justificación: Devuelve los registros de l tabla con la funcion registro del modulo tabla.

---

icantidadDeAccesos(**in** t_1 : `string` **in** a : `archibase`) → res : nat

1: $res \leftarrow cantidadDeAccesos(dameTabla(t_1))$ ▷ O(1)

Complejidad: O(1)
Justificación: Devuelve la cantidad de accesos a un tabla con la funcion cantiadDeAccesos el modulo tabla.

5 DiccionarioLog(κ,σ)

5.1 Especificación

Interfaz

parámetros formales
géneros σ, κ
función COPIAR(**in** s: σ) $\rightarrow$ res : σ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} s$}
Complejidad: $O(copy(k))$
Descripción: función de copia de σ's

función $\bullet = \bullet$(**in** s_1, s_2: σ) $\rightarrow$ res : σ
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ $s_1 = s_2$}
Complejidad: $O(equal(s_1, s_2))$

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

función $\bullet = \bullet$(**in** s_1, s_2: κ) $\rightarrow$ res : κ
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ $s_1 = s_2$}
Complejidad: $O(equal(s_1, s_2))$

se explica con: DICCIONARIO(κ,σ), ITERADOR UNIDIRECCIONAL MODIFICABLE (TUPLA(κ,σ)).

géneros: `Diccionario(`κ`,`σ`)`, `diccLog(`σ`)` , `itdiccLog(`σ`)`

Trabajo Práctico 2Operaciones básicas de DiccLog

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

ESVACIO?(**in** d: `diccLog(`κ`,`σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ (claves(d) = $\emptyset$)}
Complejidad: O(m)
Descripción: Devuelve verdadero sii la clave de entrada esta definida en el diccionario.

DEFINIR(**in**/**out** d: `diccLog(`κ`,`σ`)`, **in** c: κ, **in** s: σ)
Pre $\equiv$ { $\neg$ def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d $=_{\text{obs}}$ definir(c,s,d_0)}
Complejidad: O(log(m))
Descripción: Agrega un elemento al diccionario.

DEFINIDO?(**in** d: `diccLog(`κ`,`σ`)`, **in** c: κ) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ def?(c,d)}
Complejidad: O(log(m))
Descripción: Devuelve verdadero sii la clave de entrada esta definida en el diccionario.

OBTENER(**in**/**out** d: `diccLog(`κ`,`σ`)`, **in** c: κ) $\rightarrow$ res : σ
Pre $\equiv$ {def?(c,d) $\wedge$ d=d_0}

Post ≡ {alias(res =$_{obs}$ obtener(c,d)) ∧ (∀ c' : κ)(c' ∈ claves(d) ⇒$_{L}$ obtener(c,d) =$_{obs}$ obtener(c,d_0))}
Complejidad: O(log(m))
Descripción: Devuelve el significado de la clave dada en el diccionario
Aliasing: res es modificable

BORRAR(**in/out** d: `diccLog(`κ`,`σ`)`, **in** c: κ)
Pre ≡ {def?(c,d) ∧ d=d_0}
Post ≡ {d =$_{obs}$ borrar(c,d_0)}
Complejidad: O(log(m))
Descripción: Borra el elemento correspondiente a la clave c del diccionario.

MAXIMO(**in** d: `diccLog(`κ`,`σ`)`) → res : κ
Pre ≡ {¬∅?($claves(d)$)}
Post ≡ {$(\forall c : \kappa)(def?(c,d) \Rightarrow_{L} c \leq res)$}
Complejidad: O(log(m))
Descripción: Devuelve la clave de máximo valor

MINIMO(**in** d: `diccLog(`κ`,`σ`)`) → res : κ
Pre ≡ {¬∅?($claves(d)$)}
Post ≡ {$(\forall c : \kappa)(def?(c,d) \Rightarrow_{L} c \geq res)$}
Complejidad: O(log(m))
Descripción: Devuelve la clave de mínimo valor

Trabajo Práctico 2Operaciones básicas de ItLog

CREARIT(**in** a: `diccLog(`σ`)`) → res : `ItLog(`σ`)`
Pre ≡ {true}
Post ≡ {alias(esPermutación(SecuSuby(res)= a))}
Complejidad: O(1)
Descripción: Creo un iterador al primer elemento del diccionario
Aliasing: El iterador se invalida sy y solo si se elimina el elemento siguiente del iterador sin ultilizar la funcion EliminarSiguiente.

HAYSIGUIENTE(**in** i: `ItLog(`κ`,`σ`)`) → res : `bool`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ haySiguiente?(i)}
Complejidad: O(1)
Descripción: Confirma si existe un elemento adelante

SIGUIENTECLAVE(**in** i: `ItLog(`κ`,`σ`)`) → res : κ
Pre ≡ {haySiguiente?(i)}
Post ≡ {res =$_{obs}$ Π_1(Siguiente(i))}
Complejidad: O(1)
Descripción: Devuelve la clave siguiente a la posición del iterador
Aliasing: res es modificable si y solo si i es modificable

SIGUIENTESIGNIFICADO(**in** i: `ItLog(`κ`,`σ`)`) → res : σ
Pre ≡ {haySiguiente?(i)}
Post ≡ {res =$_{obs}$ Π_2(Siguiente(i))}
Complejidad: O(1)
Descripción: Devuelve el significado siguiente a la posición del iterador
Aliasing: res es modificable si y solo si i es modificable

BUSCAR(**in** d: `diccLog(`κ`,`σ`)`, **in** c: κ) → res : `ItLog(`σ`)`
Pre ≡ {}
Post ≡ {**if** def?(c, a) **then** Π_1(Siguiente(res))=$_{obs}$ c **else** vacia?(Siguientes(res)) **fi**}
Complejidad: O(log(m))
Descripción: Devuelve el iterador con siguiente en el nodo de clave c. Si no está definida, el iterador se encuentra en el lugar donde el elemento debería agregarse y avanzarlo costará $O(m)$ donde m es la cantidad de claves definidas

ELIMINARSIGUIENTE(**in/out** i: `ItLog(`κ`,`σ`)`)

Pre $\equiv$ {i = i_0 $\wedge$ haySiguiente?(i)}
Post $\equiv$ {i $=_{obs}$ EliminarSiguiente(i_0) }
Complejidad: O($log(m)$)
Descripción: Elimina la clave siguiente al iterador junto con su significado. m es la cantidad de claves definidas

AVANZAR(**in/out** i: `ItLog(`κ`,`σ`)`)
Pre $\equiv$ {i_0 $=_{obs}$ i $\wedge$ haySiguiente?(i)}
Post $\equiv$ {i$=_{obs}$ Avanzar(i_0)}
Complejidad: O(m)
Descripción: Avanza el iterador. En caso de haber utilizado Buscar previamente, la complejidad es lineal con la cantidad de elementos definidos m. En cualquier otro caso, es O(1)

5.2 Pautas de implementación

Representación

5.2.1 Justificación

`diccLog(`κ`, `σ`)` **se representa con** `puntero(nodoAB(`κ`, `σ`)))`

donde `nodoAB` es `tupla(`*padre*: `puntero(nodoAB(`κ`, `σ`))` ,
clave: κ ,
significado: σ ,
izq: `puntero(nodoAB(`κ`, `σ`))` ,
der: `puntero(nodoAB(`κ`, `σ`))`)

5.2.2 Invariante de representación

Informal
(1) La raíz no tiene padre
(2) Si un nodo está conectado al árbol, sus hijos lo tienen como padre. Además, la clave de un hijo izquierdo es menor que la clave de su padre y la clave de un hijo derecho es mayor que la de su padre.
Formal

Rep : puntero(nodoAB) p $\longrightarrow$ bool
Rep(p) $\equiv$ true $\iff$ (1) $p \rightarrow padre = NULL$ $\wedge$
(2) $(\forall p_1 : puntero(nodoAB))(conectado(p, p_1) \Rightarrow_L ((p \rightarrow izq \neq NULL \Rightarrow_L (p_1 \rightarrow clave > p1 \rightarrow izq \rightarrow clave \wedge p_1 \rightarrow izq \rightarrow padre = p_1)) \wedge (p_1 \rightarrow der \neq NULL \Rightarrow_L (p_1 \rightarrow clave < p_1 \rightarrow der \rightarrow clave \wedge p_1 \rightarrow der \rightarrow padre = p_1)))$

conectado : puntero(nodoAB) $\times$ puntero(nodoAB) $\longrightarrow$ bool
conectado(p_1,p_2) $\equiv$ **if** $p_1 = NULL$ **then**
false
else
$p_1 = p_2 \vee$ $conectado(p_1 \rightarrow izq, p_2) \vee$ $conectado(p_1 \rightarrow der, p_2)$
fi

5.2.3 Predicado de abstracción

Abs : puntero(nodoAB) p $\longrightarrow$ dicc(κ, σ) {Rep(p)}
Abs(p) $\equiv$ $d : dicc(\kappa, \sigma)$ | $(\forall c : \kappa)(def?(c, d) =_{obs} EstaEnUnNodo?(c, p) \wedge_L obtener(c, d) =_{obs} DameSig(c, p))$
EstaEnUnNodo? : $\kappa \times$ puntero(nodoAB) $\longrightarrow$ bool
EstaEnUnNodo?(c,p) $\equiv$ **if** $p = NULL$ **then**
false
else
$(p \rightarrow clave = c) \vee (EstaEnUnNodo?(c, p \rightarrow izq) \vee EstaEnUnNodo?(c, p \rightarrow der)$
fi

DameSig : $\kappa \times$ puntero(nodoAB) $\longrightarrow \sigma$
DameSig(c,p) $\equiv$ **if** $p \rightarrow clave = c$ **then** $p \rightarrow significado$ **else** (**if** EstaEnUnNodo?(c, p→der) **then** DameSig(c, p→der) **else** DameSig(c, p→izq) **fi**) **fi**

Trabajo Práctico 2Representación del Iterador

Como todo iterador, ItLog(κ,σ) posee un puntero al siguiente elemento, pero además guarda en todo momento una referencia a su padre para facilitar el borrado del siguiente y la reencadenación de los nodos, pero ante todo permite realizar la operación AgregarComoSiguiente. Dadas las restricciones de esta operación, era necesario que no hubiera siguiente nodo previo al agregado pero sin olvidar quien será su padre.

Por otro lado, el recorrido permite tener seguimiento de los nodos que ya se recorrieron y avanzar usando el método DFS. Pero como se desea poder buscar en $O(log(m))$ (donde m es la cantidad de claves), se usa el elemento *busca* para distinguir si un iterador llego a donde está a través del método DFS (una sucesión de Avanzar) o utilizando la función Buscar.

`ItLog(`κ`,`σ`)` **se representa con** `iter`

donde `iter` es `tupla`(*siguiente*: `puntero(nodoAB(`κ`,`σ`))` ,
anterior: `puntero(nodoAB(`κ`,`σ`))` ,
recorrido: `pila(puntero(NodoAB(`κ`,`σ`))` ,
busca: `bool`)

Informal

(1) *i.anterior* es el padre de *i.siguiente*
(2) *i.recorrido* es vacío o contiene la secuencia de elementos que falta recorrer, esten definidos o no
(3) Si *i.busca* es true, entonces *i.recorrido* está vacía

Formal

Rep : iter $\longrightarrow$ bool
Rep(i) $\equiv$ true $\iff$ (1) (i.siguiente=NULL $\vee_L$ i.siguiente→padre=i.anterior) $\wedge$ (i.anterior=NULL $\vee_L$ i.anterior→der=i.siguiente $\vee$ i.anterior→izq=i.siguiente) $\wedge_L$
(2) (i.siguiente=NULL $\vee_L$ i.recorrido=Aplilados(i.siguiente, Odin(i.siguiente), vacia)) $\wedge$
(3) $i.busca \rightarrow EsVacia?(i.recorrido)$

Abs : iter i $\longrightarrow$ ItMod(σ) {Rep(i)}
Abs(i) $\equiv$ a: ItMod(σ) | Siguientes(a)=$_{obs}$Desde(i.siguiente→clave,preoderABB(Odin(i.siguiente))) $\wedge$
Anteriores(a)=$_{obs}$Hasta(i.siguiente→clave,preoderABB(Odin(i.siguiente)))

preorderABB : puntero(nodoAB(κ;σ)) $\longrightarrow$ secu(tupla<κ;σ>)
preorderABB(p) $\equiv$ **if** p=NULL **then**
<>
else
(<p→clave, p→ significado> • preorderABB(p →izq))&(preorderABB(p →der))
fi

Desde : κ $c \times$ secu(tupla<κ;σ>) s $\longrightarrow$ secu(tupla<κ, σ>) {Esta?(c, s)}
Desde(c,s) $\equiv$ **if** Π_1(prim(s))=c **then** s **else** Desde(c, fin(s)) **fi**

Hasta : κ $c \times$ secu(tupla<κ;σ>) s $\longrightarrow$ secu(tupla<κ, σ>) {Esta?(c, s)}
Hasta(c,s) $\equiv$ **if** Π_1(prim(s))=c **then** com(s) **else** Hasta(c, com(s)) **fi**

Odin : puntero(nodoAB(κ;σ)) n $\longrightarrow$ puntero(nodoAB(κ;σ)) {$n \neq NULL$}
Odin(n) $\equiv$ **if** n →padre=NULL **then** n **else** Odin($n \rightarrow padre$) **fi**

Apliados : puntero(nodoAB(κ;σ)) $\times$ puntero(nodoAB(κ;σ)) $\times$ pila(puntero(nodoAB(κ;σ))) $\longrightarrow$
pila(puntero(nodoAB(κ;σ)))

Apilados(n, p, ps) $\equiv$ **if** p=n **then**
ps
else
if $p \rightarrow der$=NULL **then**
if $p \rightarrow izq$=NULL **then**
Apilados(n, tope(ps), desapilar(ps))
else
Apliados(n, $p \rightarrow izq$, apilar($p \rightarrow izq$, desapilar(ps)))
fi
else
if $p \rightarrow izq$=NULL **then**
Apliados(n, tope(ps), apilar($p \rightarrow der$,desapilar(ps)))
else
Apilados(n, $p \rightarrow izq$, apilar($p \rightarrow izq$, apliar($p \rightarrow der$, desapilar(ps))))
fi
fi
fi

Trabajo Práctico 2Operaciones privadas del iterador

EliminarHoja(**in/out** i: `ItLog(`σ`)`)
Pre $\equiv$ {i = i_0 $\wedge$ haySiguiente?(i) $\wedge_L$ (Siguiente(i).der=NULL $\wedge$ Siguiente(i).izq=NULL)}
Post $\equiv$ {i $=_{obs}$ EliminarSiguiente(i_0) }
Complejidad: O(1)
Descripción: Elimina la siguiente hoja del iterador

EliminarRaiz(**in/out** i: `ItLog(`σ`)`)
Pre $\equiv$ {i = i_0 $\wedge$ haySiguiente?(i) $\wedge_L$ Siguiente(i).padre=NULL}
Post $\equiv$ {i $=_{obs}$ EliminarSiguiente(i_0) }
Complejidad: O(1)
Descripción: Elimina la siguiente hoja del iterador

EliminarConUnHijo(**in/out** i: `ItLog(`σ`)`)
Pre $\equiv$ {i = i_0 $\wedge$ haySiguiente?(i) $\wedge_L$ ((Siguiente(i).der=NULL $\wedge$ Siguiente(i).izq$\neq$NULL) $\vee$ (Siguiente(i).der$\neq$NULL $\wedge$ Siguiente(i).izq=NULL))}
Post $\equiv$ {i $=_{obs}$ EliminarSiguiente(i_0) }
Complejidad: O(1)
Descripción: Elimina el siguiente nodo con un solo hijo del iterador

AgregarComoSiguiente(**in/out** i: `ItLog(`κ`,`σ`)`, **in** c: κ, **in** s: σ)
Pre $\equiv$ {i = i_0 $\wedge$ PuedoAgregarloAhi?(c,i)}
Post $\equiv$ {i $=_{obs}$ AgregarComoSiguiente(i_0,<c,s>) }
Complejidad: O(1)
Descripción: Define la clave c con significado s

5.3 Algoritmos

Algoritmos

Trabajo Práctico 2Algoritmos del módulo

---

iVacio() $\rightarrow$ res : puntero(nodoAB(κ, σ))
1: $res \leftarrow NULL$ ▷ $O(1)$

Complejidad: $O(1)$

---

iEsVacio?(**in** p: `puntero(nodoAB(`κ`, `σ`))` $\rightarrow$ res : bool

1: $res \leftarrow p = NULL$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Hacer una operacion booleana.

iDefinir(**in/out** p: `puntero(nodoAB(`κ`, `σ`))` , **in** c: κ , **in** s: σ)

1: **if** $(esVacio?(p))$ **then**
2: $t \leftarrow < NULL, c, s, NULL, NULL >$
3: **else**
4: $AgregarComoSiguiente(Buscar(p, n), n, s)$ ▷ $O(log(m))$
5: **end if**

Complejidad: $O(log(m))$
Justificación: Como va recorriendo el arbol de izquierda a derecha , como maximo va recorrer la altura del arbol que es log(m) donde m es la cantidad de nodos. Ademas como los nodos se distribuyen de manera uniforme , siempre tendre un arbol balanceado (es decir no voy a tener un arbol degenerado hacia izq. o der.).

iDefinido?(**in** p: `puntero(nodoAB(`κ`, `σ`))` , **in** c: κ)$\rightarrow$ res : bool

1: $res \leftarrow HaySiguiente(Buscar(p, c))$

Complejidad: O$(log(m))$
Justificación: El peor caso es recorrer la altura del arbol, que por distribución uniforme es $O(log(m))$.

iObtener(**in** p: `puntero(nodoAB(`κ`, `σ`))` , **in** n: κ)$\rightarrow$ res: σ

1: $res \leftarrow (SiguienteSignificado(Buscar(p, n))$ ▷ O(log(m))

Complejidad: O(log(m))
Justificación: Asumiendo una distribución uniforme, la altura del arbol es el orden $log(m)$ y el peor caso es tener que bajar hasta las hojas.

iBorrar(**in/out** p: `puntero(nodoAB(`κ`, `σ`))` , **in** n: κ)

1: $EliminarSiguente(Buscar(p, n))$ ▷ $O(log(m))$

Complejidad: $O(log(m))$
Justificación: Gracias a la distribución uniforme de los datos, buscar el nodo con la clave n cuesta O(log(m)) (con m la cantidad de nodos) y eliminarlo es O(log(m)).

iMinimo(**in/out** p: `puntero(nodoAB(`κ`, `σ`))`$\rightarrow$ res : κ

1: $puntero(nodoAB(\kappa, \sigma))$ $a \leftarrow p$ ▷ $O(1)$
2: **while** $a \rightarrow izq \neq NULL$ **do** ▷ A lo sumo recorro la altura del arbol // $O(log(m))$
3: $a \leftarrow a \rightarrow izq$ ▷ $O(1)$
4: **end while**
5: $res \leftarrow a \rightarrow clave$ ▷ $O(1)$

Complejidad: $O(log(m))$
Justificación: Gracias a la distribución uniforme de los datos, la altura del árbol es $log(m)$.

iMaximo(**in/out** p: `puntero(nodoAB(`κ`, `σ`))`$\rightarrow res : \kappa$

1: $puntero(nodoAB(\kappa,\sigma))\ a \leftarrow p$ ▷ $O(1)$
2: **while** $a \rightarrow der \neq NULL$ **do** ▷ A lo sumo recorro la altura del arbol // $O(log(m))$
3: $a \leftarrow a \rightarrow der$ ▷ $O(1)$
4: **end while**
5: $res \leftarrow a \rightarrow clave$ ▷ $O(1)$

Complejidad: $O(log(m))$
Justificación: Gracias a la distribución uniforme de los datos, la altura del árbol es $log(m)$.

Trabajo Práctico 2Algoritmos del iterador

iCrearIt(**in** p: `puntero(nodoAB(`κ`, `σ`))`$\rightarrow res$: iter

1: $res \leftarrow < p, NULL, Vacia(), false >$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: La función Vacia es la del módulo Pila(α) del apunte.

iHaySiguiente(**in** i: `iter`)$\rightarrow res$: bool

1: $res \leftarrow i.siguiente \neq NULL$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: La función EsVacia? es la del módulo Pila(α) del apunte.

iSiguienteClave(**in** i: `iter`)$\rightarrow res : \kappa$

1: $res \leftarrow (i.siguiente \rightarrow clave)$ ▷ Genero aliasing con el significado // $O(1)$

Complejidad: $O(1)$

iSiguienteSignificado(**in** i: `iter`)$\rightarrow res : \sigma$

1: $res \leftarrow (i.siguiente \rightarrow significado)$ ▷ Genero aliasing con el significado // $O(1)$

Complejidad: $O(copy(\sigma))$

iBuscar(**in/out** p: `puntero(nodoAB(`κ`, `σ`))`, **in** c: κ)$\rightarrow res$:iter

1: $res \leftarrow CrearIt(p)$ ▷ $O(1)$
2: $res.busca \leftarrow true$
3: **while** $(HaySiguiente(res) \wedge SiguienteClave(res) \neq c)$ **do** ▷ $O(log(m))$
4: $res.anterior \leftarrow res.siguiente$
5: **if** $SiguienteClave(res) < c$ **then**
6: $(res.siguiente) \leftarrow (res.siguiente \rightarrow der)$ ▷ $O(1)$
7: **else**
8: $(res.siguiente) \leftarrow (res.siguiente \rightarrow izq)$ ▷ $O(1)$
9: **end if**
10: **end while**

Complejidad: $O(log(m))$
Justificación: Dado que la distribución es uniforme, la altura del árbol es del orden de $log(m)$ (con m la cantidad de nodos)y el peor caso sería recorrer hasta una hoja.

iEliminarHoja(**in/out** i : `iter`)

```
1: if (¬EsVacia?(i.recorrido)) then                                   ▷ O(1)
2:     i.siguiente ← Tope(i.recorrido)   ▷ Copiado de puntero es constante // O(1)
3:     i.anterior ← i.siguiente → padre                               ▷ O(1)
4:     Desapilar(i.recorrido)                                         ▷ O(1)
5: end if
```

Complejidad: $O(1)$
Justificación: Las funciones Tope, Apilar y Desapilar son las del módulo Pila(α) del apunte.

iEliminarRaiz(**in/out** i : `iter`)

```
1: if (i.siguiente.der ≠ NULL ∧ i.siguiente.izq ≠ NULL) then          ▷ O(1)
2:     (i.siguiente → izq).padre ← i.siguiente → der                  ▷ O(1)
3:     (i.siguiente → der).izq ← i.siguiente → izq                    ▷ O(1)
4:     (i.siguiente) ← i.siguiente → der                              ▷ O(1)
5: else
6:     if i.siguiente.der ≠ NULL then                                 ▷ O(1)
7:         (i.siguiente) ← i.siguiente → der                          ▷ O(1)
8:     else
9:         (i.siguiente) ← i.siguiente → izq                          ▷ O(1)
10:    end if
11: end if
```

Complejidad: $O(1)$
Justificación: Como las asignaciones de punteros son $O(1)$ y este algoritmo no hace más que eso, resulta $O(1)$

iEliminarConUnHijo(**in/out** i : `iter`)

```
puntero(nodoAB(κ, σ)) temp ← i.siguiente   ▷ Guardo un puntero al nodo que quiero borrar // O(1)
if i.siguiente → der ≠ NULL then                                      ▷ O(1)
    (temp → izq).padre ← i.anterior                                   ▷ O(1)
    if i.anterior → der = i.siguiente then
        (i.anterior → der) ← (temp → izq)                             ▷ O(1)
    else
        (i.anterior → izq) ← (temp → izq)                             ▷ O(1)
    end if
    (i.siguiente) ← temp → izq
else
    if i.anterior → der = i.siguiente then                            ▷ O(1)
        (i.anterior → der) ← (temp → der)                             ▷ O(1)
    else
        (i.anterior → izq) ← (temp → der)                             ▷ O(1)
    end if
    (i.siguiente) ← temp → der                                        ▷ O(1)
end if
temp ← NULL                                          ▷ Borro el nodo //O(1)
```

Complejidad: $O(1)$
Justificación: Este algoritmo solo asigna punteros, cuya complejidad es $O(1)$

EliminarSiguiente(**in/out** i : `iter`)

if $(i.siguiente.der \neq NULL \wedge i.siguiente.izq \neq NULL)$ **then** ▷ Si es una hoja, lo borramos de una // $O(1)$
  $EliminarHoja(i)$ ▷ $O(1)$
else ▷ $O(log(m))$
  if $(i.anterior \neq NULL)$ **then** ▷ Nos aseguramos de que no sea la raíz // $O(log(m))$
    if $\big((i.siguiente.der = NULL \wedge i.siguiente.izq \neq NULL) \vee (i.siguiente.der \neq NULL \wedge i.siguiente.izq = NULL)\big)$ **then** ▷ Caso en que tiene un solo hijo // $O(1)$
      $EliminarConUnHijo(i)$ ▷ $O(1)$
    else
      $puntero(nodoAB(\kappa, \sigma))\ temp \leftarrow (i.siguiente)$ ▷ Guardo un puntero al nodo que quiero borrar // $O(1)$
      $puntero(nodoAB(\kappa, \sigma))\ rec \leftarrow (temp \rightarrow der)$ ▷ $O(1)$
      while $(rec.izq \neq NULL)$ **do** ▷ Recorre a lo sumo la altura del arbol // $O(log(m))$
        $rec \leftarrow (rec \rightarrow izq)$ ▷ $O(1)$
      end while
      $(temp \rightarrow clave) \leftarrow (rec \rightarrow clave)$ ▷ $O(1)$
      $(temp \rightarrow significado) \leftarrow (rec \rightarrow significado)$ ▷ $O(1)$
      if $rec \rightarrow der = NULL$ **then** ▷ $O(log(m))$
        $EliminarHoja(Buscar(*i.abb, rec \rightarrow clave))$ ▷ $O(log(m))$
      else
        $EliminarConUnHijo(Buscar(*i.abb, rec \rightarrow clave))$ ▷ $O(log(m))$
      end if
      $temp \leftarrow NULL$ ▷ Borro el nodo $O(1)$
    end if
  else
    $EliminarRaiz(i)$ ▷ $O(1)$
  end if
end if

Complejidad: $O(log(m))$
Justificación: Por la distribucion uniforme de los datos, la altura del arbol es de orden $log(m)$.

---

iAgregarComoSiguiente(**in/out** i : `iter`, **in** c : κ, **in** s : σ)

1: $nodoAB(\kappa, \sigma)n \leftarrow < i.anterior, c, s, NULL, NULL >$ ▷ $O(copy(\kappa) + copy(\sigma))$
2: **if** $i.anterior \rightarrow clave > c$ **then** $(i.anterior \rightarrow der) \leftarrow \&n$ **else** $(i.anterior \rightarrow izq) \leftarrow \&n$ **fi** ▷ $O(1)$
3: $i.siguiente \leftarrow \&n$ ▷ $O(copy(\kappa) + copy(\sigma))$

Complejidad: $O(1)$

iAvanzar(**in/out** i : `iter`)

1: **if** $i.busca$ **then** ▷ Resuelve el caso patológico en que se utilizó Buscar previo a llamar a la función, armando de cero el recorrido // $O(m) + O(log(m)) = O(m)$
2: $iter\ it \leftarrow i$ ▷ $O(1)$
3: **while** $(it.anterior \neq NULL)$ **do** ▷ En peor caso recorre la altura del arbol // $O(log(m))$
4: $it.siguiente \leftarrow it.anterior$ ▷ $O(1)$
5: $it.anterior \leftarrow it.anterior \rightarrow padre$ ▷ $O(1)$
6: **end while**
7: **while** $(it.siguiente \rightarrow clave \neq i.siguiente \rightarrow clave)$ **do** ▷ A lo sumo recorre todos los elementos // $O(m)$
8: **if** $(it.siguiente \rightarrow der \neq NULL)$ **then** ▷ $O(1)$
9: $Apilar(it.recorrido, it.siguiente \rightarrow der)$ ▷ El costo de copiado de puntero constante // $O(1)$
10: **end if**
11: **if** $(it.siguiente \rightarrow izq \neq NULL)$ **then** ▷ $O(1)$
12: $Apilar(it.recorrido, it.siguiente \rightarrow izq)$ ▷ El costo de copiado de puntero constante // $O(1)$
13: **end if**
14: $it.siguiente \leftarrow Tope(it.recorrido)$ ▷ $O(1)$
15: $it.anterior \leftarrow it.siguiente \rightarrow padre$ ▷ $O(1)$
16: $Desapilar(it.recorrido)$ ▷ $O(1)$
17: **end while**
18: $i \leftarrow it$ ▷ $O(1)$
19: $i.busca \leftarrow false$ ▷ $O(1)$
20: **end if**
21: **if** $(i.siguiente \rightarrow der \neq NULL)$ **then** ▷ $O(1)$
22: $Apilar(i.recorrido, i.siguiente \rightarrow der)$ ▷ El costo de copiado de puntero constante // $O(1)$
23: **end if**
24: **if** $(i.siguiente \rightarrow izq \neq NULL)$ **then** ▷ $O(1)$
25: $Apilar(i.recorrido, i.siguiente \rightarrow izq)$ ▷ El costo de copiado de puntero constante // $O(1)$
26: **end if**
27: **if** $EsVacia?(i.recorrido)$ **then**
28: $i.anterior \leftarrow i.siguiente$ ▷ $O(1)$
29: $i.siguiente \leftarrow NULL$ ▷ $O(1)$
30: **else**
31: $i.siguiente \leftarrow Tope(i.recorrido)$ ▷ $O(1)$
32: $i.anterior \leftarrow i.siguiente \rightarrow padre$ ▷ $O(1)$
33: $Desapilar(i.recorrido)$ ▷ $O(1)$
34: **end if**

Complejidad: $O(m)$ tras usar Buscar, $O(1)$ en cualquier otro caso
Justificación: Las funciones Tope, Apilar y Desapilar son las del módulo Pila(α) del apunte. Esta operación tiene complejidad lineal cuando el iterador se encuentra en algún nodo sin tener definido su recorrido, lo cual solo puede ocurrir luego de utilizar la función Buscar (lo cual es indicado por $i.busca$).

6 DiccionarioString(σ)

6.1 Interfaz

Interfaz

parámetros formales
géneros σ
función CopiarSignificado(**in** s: σ) $\rightarrow$ res : σ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} s$}
Complejidad: $O(copy(s))$
Descripción: función de copia de σ's

función $\bullet = \bullet$(**in** s_1, s_2: σ) $\rightarrow$ res : σ
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ $s_1 = s_2$}
Complejidad: $O(equal(s_1, s_2))$

se explica con: Diccionario(string,σ), Iterador Bidireccional(tupla(string,σ)).

géneros: `Diccionario(string,`σ`)`, `diccString(`σ`)` , `ItStr(`σ`)`

Trabajo Práctico 2Operaciones básicas DiccionarioString

Vacio() $\rightarrow$ res : `diccString(`σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ vacio}
Complejidad: O(1)
Descripción: Crea un diccionario vacio.

EsVacio?(**in** d: `diccString(`σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ $\emptyset?(claves(d))$}
Complejidad: O(1)
Descripción: Comprueba si un diccionario d está vacío

Definir(**in/out** d: `diccString(`σ`)`, **in** c: `string`, **in** s: σ)
Pre $\equiv$ { $\neg$ def?(c,d) $\wedge$ d=d_0}
Post $\equiv$ {d $=_{\text{obs}}$ definir(c, s, d_0)}
Complejidad: O(L + copy(s))
Descripción: Agrega un elemento al diccionario, donde L es la longitud del string más largo definido en el diccionario d

Definido?(**in** d: `diccString(`σ`)`, **in** c: `string`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ def?(c,d)}
Complejidad: O(L)
Descripción: Devuelve verdadero sii el string de entrada c esta definida en el diccionario, donde L es la longitud del string más largo definido en el diccionario d

Obtener(**in/out** d: `diccString(`σ`)`, **in** c: `string`) $\rightarrow$ res : σ
Pre $\equiv$ {def?(c,d)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ obtener(c,d))}
Complejidad: O(L)
Descripción: Devuelve el significado de la clave dada en el diccionario, , donde L es la longitud del string más largo definido en el diccionario d
Aliasing: res es modificable si y solo si d es modificable

Borrar(**in/out** d: `diccString(`σ`)`, **in** c: `string`)
Pre $\equiv$ {def?(c,d) $\wedge$ d=d_0}

Post $\equiv$ {d $=_{\text{obs}}$ borrar(c,d_0)}
Complejidad: O(L)
Descripción: Borra el elemento correspondiente a la clave c del diccionario, donde L es la longitud del string más largo definido en el diccionario d

Maximo(**in** d: `diccLog(`κ`,`σ`)`) $\rightarrow$ res : κ
Pre $\equiv$ {$\neg\emptyset?(claves(d))$}
Post $\equiv$ {$(\forall c : \kappa)(def?(c,d) \Rightarrow_{\text{L}} c \leq res)$}
Complejidad: O(log(m))
Descripción: Devuelve la clave de máximo valor

Minimo(**in** d: `diccLog(`κ`,`σ`)`) $\rightarrow$ res : κ
Pre $\equiv$ {$\neg\emptyset?(claves(d))$}
Post $\equiv$ {$(\forall c : \kappa)(def?(c,d) \Rightarrow_{\text{L}} c \geq res)$}
Complejidad: O(log(m))
Descripción: Devuelve la clave de mínimo valor

Trabajo Práctico 2Operaciones básicas del iterador

CrearIt(**in** a: `diccString(`σ`)`) $\rightarrow$ res : `ItStr(`σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutación(SecuSuby(res)= a))}
Complejidad: O(1)
Descripción: Crea un iterador apuntando al primer elemento del diccionario
Aliasing: El iterador se invalida si y solo si se elimina el elemento siguiente del iterador sin ultilizar la funcion EliminarSiguiente.

HaySiguiente(**in** i: `ItStr(`σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ haySiguiente?(i)}
Complejidad: O(1)
Descripción: Confirma si existe un elemento adelante

SiguienteClave(**in** i: `ItStr(`σ`)`) $\rightarrow$ res : `string`
Pre $\equiv$ {haySiguiente?(i)}
Post $\equiv$ {$res =_{\text{obs}} \Pi_1$(Siguiente(i))}
Complejidad: O(1)
Descripción: Devuelve la clave siguiente a la posición del iterador
Aliasing: res es modificable si y solo si i es modificable

SiguienteSignificado(**in** i: `ItStr(`σ`)`) $\rightarrow$ res : σ
Pre $\equiv$ {haySiguiente?(i)}
Post $\equiv$ {$res =_{\text{obs}} \Pi_2$(Siguiente(i))}
Complejidad: O(1)
Descripción: Devuelve el significado siguiente a la posición del iterador
Aliasing: res es modificable si y solo si i es modificable

Buscar(**in** d: `diccString(`σ`)`, **in** c: `string`) $\rightarrow$ res : `ItStr(`σ`)`
Pre $\equiv$ {}
Post $\equiv$ {**if** def?(c,a) **then** Π_1(Siguiente(res))$=_{\text{obs}}$ c **else** vacia?(Siguientes(res)) **fi**}
Complejidad: O(L)
Descripción: Devuelve el iterador con siguiente en el nodo de clave c. Si no está definida, el iterador se encuentra en el lugar donde el elemento debería agregarse y avanzarlo costará $O(n)$ con n la cantidad de claves definidas

EliminarSiguiente(**in/out** i: `ItStr(`σ`)`)
Pre $\equiv$ {i = i_0 $\wedge$ haySiguiente?(i)}
Post $\equiv$ {i $=_{\text{obs}}$ EliminarSiguiente(i_0) }
Complejidad: O(1)
Descripción: Elimina la clave siguiente al iterador junto con su significado

Avanzar(**in/out** i: `ItStr(`σ`)`)

Pre $\equiv$ {$i_0 =_{obs}$ i $\wedge$ haySiguiente?(i)}
Post $\equiv$ {i$=_{obs}$ Avanzar(i_0)}
Complejidad: O(n)
Descripción: Avanza el iterador. En caso de haber utilizado Buscar previamente, la complejidad es lineal con la cantidad de elementos definidos n. En cualquier otro caso, es O(1)

6.2 Pautas de implementacion

Representación Trabajo Práctico 2Representación del diccionario

Para aprovechar al máximo el hecho de que las claves sean *strings*, se implementó el diccionario sobre un trie, donde cada nodo tiene hasta 256 hijos, representando su posición la letra que se agrega a la clave de su padre (utilizando la función *ord*). Sin embargo, los nodos no guardan la clave que tienen asociada con el objetivo de ahorrar memoria y reducir la complejidad del agregado. Esto es trabajo del iterador.

`diccString(`σ`)` **se representa con** `puntero(nodo(`σ`))`

donde `nodo(`σ`)` es `tupla(`*significado*: `puntero(`σ`)` ,
caracteres: `arreglo[256] de puntero(nodo(`σ`))` ,
padre: `puntero(nodo(`σ`))`)

6.2.1 Invariante de representación

Informal

(1) El *puntero*(*nodo*(σ)) que representa al arbol no tiene padre (es la raíz)
(2) Los hijos de cada nodo (a donde apuntan los punteros del arreglo) lo tienen como padre.

Formal
Rep : puntero(nodoAB) $p \longrightarrow$ bool
Rep(p) $\equiv$ true $\Longleftrightarrow$ (1) $p \rightarrow padre = NULL \wedge$
(2) $(\forall p_1 : puntero(nodo))\Big(EsDescendiente?(p_1, p) \Rightarrow_{\text{L}} (\forall p_2 : puntero(nodo))\big((p_2 \neq NULL \wedge_{\text{L}} EsHijo?(p_2, p_1, 255)) \Rightarrow_{\text{L}} (p_2 \rightarrow padre = p_1)\big)\Big)$

EsHijo? : puntero(nodo) $\times$ puntero(nodo) $\times$ nat $\longrightarrow$ bool {n<256}
EsHijo?(p_1,p_2,n) $\equiv$ **if** n=0 **then** false **else** $p_2 \rightarrow$caracteres[n]=$p_1 \vee$ EsHijo?(p_1,p_2,n-1) **fi**

EsDescendiente? : puntero(nodo) $\times$ puntero(nodo) $\longrightarrow$ bool
EsDescendiente?(p_1,p_2) $\equiv$ **if** p_1=NULL **then** false **else** $p_1 = p_2 \vee$ EsDescendiente?($p_1 \rightarrow padre, p_2$) **fi**

indice : puntero(nodo(σ) $\times$ puntero(nodo(σ)) $\times$ nat n $\longrightarrow$ nat {EsHijo?(p_1,p_2) $\wedge$ n<256}
indice(p_1,p_2, n) $\equiv$ **if** $p_1 \rightarrow caracteres[n] = p_2$ **then** n **else** indice($p_1, p_2, n-1$) **fi**

6.2.2 Predicado de abstracción

Abs : puntero(nodo) $p \longrightarrow$ dicc(string, σ) {Rep(p)}
Abs(p) $\equiv$ d : dicc(string,σ) | $(\forall s : string)\Big(\big(Def?(s, d) \Longleftrightarrow (encontrarPalabra(s, p) \neq NULL \wedge_{\text{L}} encontrarPalabra(s, p) \rightarrow significado \neq NULL)\big) \wedge \big(*(encontrarPalabra(s, p) \rightarrow significado) = obtener(s, p)\big)\Big)$

encontrarPalabra : string $\times$ puntero(nodo) $\longrightarrow$ puntero(nodo)
encontrarPalabra(s, p) $\equiv$ **if** $vacia(s) \vee p = NULL$ **then**
p
else
encontrarPalabra($fin(s)$, $p \rightarrow caracteres[ord(prim(s))]$)
fi

Asumo que TAD String es secu(char) con char tipo enumerado

Trabajo Práctico 2Representación del iterador

De forma similar a ItLog(κ,σ), este iterador posee un puntero al siguiente elemento, pero guarda una referencia al padre para poder realizar la operación AgregarComoSiguiente. Dadas las restricciones de esta operación, era necesario que no hubiera siguiente nodo previo al agregado pero sin olvidar quien será su padre.

Por otro lado, el recorrido permite tener seguimiento de los nodos que ya se recorrieron y avanzar usando el método DFS. Pero como se desea poder buscar en $O(L)$ (donde L es la longitud maxima de un string clave), se usa el elemento *busca* para distinguir si un iterador llego a donde está a través del método DFS (una sucesión de Avanzar) o utilizando la función Buscar.

`ItStr(`σ`)` se representa con `iter`

donde `iter` es `tupla`(*siguiente*: `puntero(nodo(`σ`))` ,
anterior: `puntero(nodo(`σ`))` ,
recorrido: `pila(data)` ,
clave: `string` ,
busca: `bool`)

donde `data` es `tupla`(*sig*: `puntero(nodo(`σ`))` , *clav*: `string`)

Invariante de representación

Informal:
(1) *i.anterior* es el padre de *i.siguiente*
(2) *i.recorrido* es vacío o contiene la secuencia de elementos que falta recorrer junto con su clave asociada, esten definidos o no
(3) Si *i.busca* es true, entonces *i.recorrido* está vacía

Formal:
Rep : iter i $\longrightarrow$ bool
Rep(i) $\equiv$ true $\iff$ (1) (i.siguiente=NULL $\vee_{\text{L}}$ i.siguiente$\rightarrow$padre=i.anterior) $\wedge$ (i.anterior=NULL $\vee_{\text{L}}$ i.anterior$\rightarrow$der=i.siguiente $\vee$ i.anterior$\rightarrow$izq=i.siguiente) $\wedge_{\text{L}}$
(3) $i.busca \rightarrow EsVacia?(i.recorrido)$

Función de abstracción

Abs : iter i $\longrightarrow$ ItMod(σ) {Rep(i)}
Abs(i) $\equiv$ a: ItMod(σ) |
Siguientes(a)=$_{\text{obs}}$Desde(clave(Odin(i.siguiente),i.siguiente, <>),preoder(Odin(i.siguiente), <>, 0)) $\wedge$
Anteriores(a)=$_{\text{obs}}$Hasta(clave(Odin(i.siguiente),i.siguiente, <>),preoder(Odin(i.siguiente), <>, 0))

preorder : puntero(nodo(σ)) $\times$ string $\times$ nat $\longrightarrow$ secu(tupla<κ;σ>)
preorder(p, s, n) $\equiv$ **if** p=NULL $\vee$ $n = 256$ **then**
<>
else
if n=0 **then**
$(\langle s, p \rightarrow significado \rangle \bullet$preorder($p \rightarrow caracteres$[n],$s \circ ord(n)$,0))&preorder($p$, s, $n+1$)
else
preorder($p \rightarrow caracteres$[n],$s \circ ord(n)$,0)&preorder(p, s, $n+1$)
fi
fi

Desde : string c $\times$ secu(tupla<κ;σ>) s $\longrightarrow$ secu(tupla<κ, σ>) {Esta?(c, s)}
Desde(c,s) $\equiv$ **if** Π_1(prim(s))=c **then** s **else** Desde(c, fin(s)) **fi**

Hasta : string c $\times$ secu(tupla<string;σ>) s $\longrightarrow$ secu(tupla<κ, σ>) {Esta?(c, s)}
Hasta(c,s) $\equiv$ **if** Π_1(prim(s))=c **then** com(s) **else** Hasta(c, com(s)) **fi**

Odin : puntero(nodo(σ)) n $\longrightarrow$ puntero(nodo(σ)) $\{n \neq NULL\}$
Odin(n) $\equiv$ **if** $n \rightarrow$padre=NULL **then** n **else** Odin($n \rightarrow padre$) **fi**

Clave : puntero(nodo(σ)) $p \times$ puntero(nodo(σ) $n \times$ string $\longrightarrow$ puntero(nodo(σ)) {EsDescendiente?(n,p)}
Clave(p,n,s) $\equiv$ **if** $n = p \vee$ vacia?(s) **then** s **else** clave(p,$n \rightarrow padre$,s) $\circ\, ord^{-1}$(indice($n \rightarrow padre, n, 255$)) **fi**

Trabajo Práctico 2Operaciones privadas del iterador

ApuntaAHoja(**in/out** i: `ItStr(`σ`)`)
Pre $\equiv$ {$i_0 =_{obs}$ i $\wedge$ haySiguiente?(i)}
Post $\equiv$ {i$=_{obs}$ Avanzar(i_0)}
Complejidad: O(n)

AgregarComoSiguiente(**in/out** i: `ItStr(`σ`)`, **in** c: `string`, **in** s: σ)
Pre $\equiv$ {i = $i_0 \wedge$ PuedoAgregarloAhi?(c,i)}
Post $\equiv$ {i $=_{obs}$ AgregarComoSiguiente(i_0,<c,s>) }
Complejidad: O(1)
Descripción: Define la clave c con significado s como siguiente del iterador

6.3 Algoritmos

Algoritmos

Trabajo Práctico 2Algoritmos del módulo
De ahora en adelante, L representa la longitud del string más largo definido en el diccionario.

---

iVacio() $\rightarrow$ res : puntero(nodo(σ)))
1: $arreglo(puntero(nodo(\sigma)))$ a[256] ▷ Inicializo un arreglo vacío de 256 elementos // $O(1)$
2: $res \leftarrow < Vacia, NULL, a, NULL >$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Usa la operacion Vacia() del módulo Vector(α) de los apuntes, asumiendo que String es Vector($Char$).

---

iEsVacio?(**in** p: `puntero(nodo(`σ`))` $\rightarrow$ res: bool
1: $res \leftarrow (p = NULL)$ ▷ $O(1)$

Complejidad: $O(1)$

---

iDefinir(**in/out** p: `puntero(nodo(`σ`))` , **in** c: `string` , **in** s: σ)
1: $AgregarComoSiguiente(Buscar(c, p), c, s)$ ▷ $O(L)$

Complejidad: $O(L)$
Justificación: Usa la operación Buscar del ItStr(σ).

---

iDefinido?(**in/out** p: `puntero(nodo(`σ`))`, **in** c: `string`) $\rightarrow$ res:bool
1: $res \leftarrow HaySiguiente(Buscar(p, c))$ ▷ $O(L)$

Complejidad: $O(L)$
Justificación: Usa la operación Buscar del ItStr(σ).

---

iObtener(**in/out** p: `puntero(nodo(`σ`))` , **in** c: `string`)$\rightarrow$ res:σ

1: $res \leftarrow SiguienteSignificado(Buscar(p, c))$ ▷ Genero un alias // $O(L)$

Complejidad: $O(L)$
Justificación: Usa la operación Buscar del ItStr(σ).

iSignificado(**in/out** p: `puntero(nodo(`σ`))` , **in** c: `string`

1: $res \leftarrow SiguienteClave(Buscar(p, c))$ ▷ Genero un alias // $O(L)$

Complejidad: $O(L)$
Justificación: Usa la operación Buscar del ItStr(σ).

iBorrar(**in/out** p: `puntero(nodo(`σ`))` , **in** c: `string`

1: $EliminarSiguiente(Buscar(p, c))$ ▷ $O(L)$

Complejidad: $O(L)$
Justificación: Usa las operaciones Buscar ($O(L)$) y EliminarSiguiente ($O(L)$) del ItStr(σ).

iMinimo(**in/out** p: `puntero(nodo(`σ`))`$\rightarrow$ res :string

1: $ItStr(\sigma)$ $i \leftarrow CrearIt(p)$ ▷ $O(1)$
2: **while** $\neg ApuntaAHoja(i)$ **do** ▷ $O(L)$
3: $Avanzar(i)$ ▷ $O(L)$
4: **end while**
5: $res \leftarrow i.clave$ ▷ $O(1)$

Complejidad: $O(L)$
Justificación: Dado el recorrido DFS del iterador sobre el trie y el hecho de que se mantiene que toda hoja tiene asociada una clave (ver iEliminar), la primer hoja que el iterador encuentre avanzando será la que esté más a la izquierda y, por lo tanto, la de menor clave asociada. Entonces, a lo sumo debe recorrer la altura del árbol.

iMaximo(**in/out** p: `puntero(nodo(`σ`))`$\rightarrow$ res :string

1: $puntero(nodo(\sigma))$ $a \leftarrow p$ ▷ $O(1)$
2: $res \leftarrow Vacia()$ ▷ $O(1)$
3: **while** $p \neq NULL$ **do** ▷ A lo sumo debo bajar la altura del árbol // $O(L)$
4: int $k \leftarrow 255$ ▷ $O(1)$
5: **while** $j > 0 \wedge a \rightarrow caracteres$[j]$= NULL$ **do** ▷ $O(256) = O(1)$
6: $j \leftarrow j - 1$ ▷ $O(1)$
7: **end while**
8: $AgregarAtras(res, ord^{-1}(j))$ ▷ $O(1)$
9: $a \leftarrow a \rightarrow caracteres$[j] ▷ $O(1)$
10: **end while**

Complejidad: $O(L)$

Trabajo Práctico 2Algoritmos del iterador

iCrearItS(**in** p: `puntero(nodo(`σ`))`$\rightarrow$ res:iter

1: $res \leftarrow < p, NULL, Vacia(), false, Vacia() >$

Complejidad: $O(1)$
Justificación: La función Vacia es la del módulo Pila(α) del apunte.

iHaySiguiente(**in** i : `iter`)→ res:bool

1: $res \leftarrow i.siguiente \neq NULL$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: La función EsVacia? es la del módulo Pila(α) del apunte.

iSiguienteClave(**in** i : `iter`)→ res:σ

1: $res \leftarrow (i.clave)$ ▷ Genero aliasing con la clave // $O(1)$

Complejidad: $O(1)$

iSiguienteSignificado(**in** i : `iter`)→ res:σ

1: $res \leftarrow (i.siguiente \rightarrow significado)$ ▷ Genero aliasing con el significado // $O(1)$

Complejidad: $O(1)$

iEliminarSiguiente(**in/out** i : `iter`)

1: $i.siguiente \rightarrow significado \leftarrow NULL$ ▷ $O(1)$
2: **while** ($ApuntaAHoja(i) \wedge i.siguiente \rightarrow significado = NULL \wedge i.anterior \neq NULL$) **do** ▷ Voy borrando todos los nodos hasta llegar a alguno con significado u otros hijos // $O(L)$
3: $int\ j \leftarrow ord(Ultimo(i.clave))$
4: $i.anterior$[j]$= NULL$ ▷ Desconecto al nodo de su padre //$O(1)$
5: $i.siguiente \leftarrow i.anterior$ ▷ $O(1)$
6: $i.anterior \leftarrow i.anterior \rightarrow padre$ ▷ $O(1)$
7: **end while**
8: **if** $EsVacia?(i.recorrido)$ **then**
9: $i.anterior \leftarrow i.siguiente$
10: $i.siguiente \leftarrow NULL$
11: **else**
12: $i.siguiente \leftarrow Tope(i.recorrido).sig$
13: $i.clave \leftarrow Tope(i.recorrido).clav$
14: $i.anterior \leftarrow i.siguiente \rightarrow padre$
15: $Desapilar(i.recorrido)$
16: **end if**

Complejidad: $O(L)$
Justificación: En peor caso debo recorrer toda la altura del árbol L.

iApuntaAHoja(**in** i : `iter`)→ res:bool

1: $res \leftarrow true$ ▷ $O(1)$
2: $int\ j \leftarrow 255$ ▷ $O(1)$
3: **while** ($j > 0 \wedge res$) **do** ▷ A lo sumo recorro los 256 hijos del nodo, pero es constante // $O(256) = O(1)$
4: $res \leftarrow i.siguiente \rightarrow recorrido$[j]$= NULL$ ▷ $O(1)$
5: $j \leftarrow j - 1$ ▷ $O(1)$
6: **end while**

Complejidad: $O(1)$

iAgregarComoSiguiente(**in/out** i: `iter`, **in** c: `string`, **in** s: σ)

```
1:  puntero(σ) p ← &s                                                          ▷ O(1)
2:  if (c = i.clave) then
3:      (i.siguiente → significado) ← p
4:  else                                          ▷ A lo sumo j avanza de 0 a L // O(L)
5:      int j ← 0
6:      while (c[j] ≠ Ultimo(i.clave) do                                       ▷ O(L)
7:          j ← j + 1                                                          ▷ O(1)
8:      end while
9:      arreglo(puntero(nodo(σ))) a[256]                                       ▷ O(1)
10:     nodo(σ) n ←< NULL, a, i.anterior >                                     ▷ O(1)
11:     i.anterior → caracteres[j] ← &n                                        ▷ O(1)
12:     i.siguiente ← n                                                        ▷ O(1)
13:     j ← j + 1                                                              ▷ O(1)
14:     while (j < Longitud(c)) do   ▷ Agrego los nodos necesarios para completar el string // O(L)
15:         arreglo(puntero(nodo(σ))) a[256]                                   ▷ O(1)
16:         nodo(σ) n ←< NULL, a, i.anterior >                                 ▷ O(1)
17:         i.anterior → caracteres[j] ← &n                                    ▷ O(1)
18:         i.anterior ← i.siguiente                                           ▷ O(1)
19:         i.siguiente ← n                                                    ▷ O(1)
20:         j ← j + 1                                                          ▷ O(1)
21:     end while
22:     i.clave ← Copiar(c)                          ▷ Costo de copiado del string c // O(L)
23:     i.siguiente → significado ← p                                          ▷ O(1)
24: end if
```

Complejidad: $O(L)$

Justificación: Entre los dos ciclos de la rama *Else*, el contador j avanzará a lo sumo entre 0 y L (la longitud del máximo string). Inevitablemente, al terminar la rama *Else* deberá pagarse el costo de copiado del string c, pues

iAvanzar(**in/out** i: `iter`)

1: **if** $i.busca$ **then** ▷ Resuelve el caso patológico en que se utilizó Buscar previo a llamar a la función, armando de cero el recorrido // $O(L) + O(n.L) = O(n.L)$
2: $iter\ it \leftarrow i$ ▷ $O(1)$
3: **while** $(it.anterior \neq NULL)$ **do** ▷ En peor caso recorre la altura del trie // $O(L)$
4: $it.siguiente \leftarrow it.anterior$ ▷ $O(1)$
5: $it.anterior \leftarrow it.anterior \rightarrow padre$ ▷ $O(1)$
6: **end while**
7: **while** $(it.clave \neq i.clave)$ **do** ▷ A lo sumo recorre todos los elementos // $O(n.L)$
8: $int\ j \leftarrow 256$
9: **while** $(j \geq 0)$ **do** ▷ $O(256.L) = O(L)$
10: **if** $i.siguiente \rightarrow caracteres$[j]$\neq NULL$ **then** ▷ $O(f(L) + L) = O(L)$ Pues $f(L) \leq L$
11: $string\ nuevaclave \leftarrow Copiar(i.clave)$ ▷ $O(L)$
12: $Apilar(i.recorrido, \langle i.siguiente \rightarrow caracteres[j], Agregar(nuevaclave, ord^{-1}(j)\rangle))$ ▷ $O(f(L))$
13: **end if**
14: **end while**
15: $it.siguiente \leftarrow Tope(it.recorrido).sig$ ▷ $O(1)$
16: $it.clave \leftarrow Copiar(Tope(it.recorrido).clav)$ ▷ $O(L)$
17: $it.anterior \leftarrow it.siguiente \rightarrow padre$ ▷ $O(1)$
18: $Desapilar(it.recorrido)$ ▷ $O(1)$
19: **end while**
20: $i \leftarrow it$ ▷ $O(1)$
21: $i.busca \leftarrow false$ ▷ $O(1)$
22: **end if**
23: $j \leftarrow 256$ ▷ $O(1)$
24: **while** $(j \geq 0)$ **do**
25: **if** $i.siguiente \rightarrow caracteres$[j]$\neq NULL$ **then** ▷ $O(f(L) + L) = O(L)$ Pues $f(L) \leq L$
26: $string\ nuevaclave \leftarrow Copiar(i.clave)$ ▷ $O(L)$
27: $Apilar(i.recorrido, \langle i.siguiente \rightarrow caracteres[j], Agregar(nuevaclave, ord^{-1}(j)\rangle))$ ▷ $O(f(L))$
28: **end if**
29: **end while**
30: **if** $EsVacia?(i.recorrido)$ **then**
31: $i.anterior \leftarrow i.siguiente$ ▷ $O(1)$
32: $i.siguiente \leftarrow NULL$ ▷ $O(1)$
33: **else**
34: **while** $(Tope(i.recorrido).sig \rightarrow significado = NULL \wedge \neg EsVacia?(i.recorrido))$ **do** ▷ En la pila no hay más nodos que en el Trie // $O(L)$
35: $Desapilar(i.recorrido)$ ▷ $O(1)$
36: **end while**
37: **if** $EsVacia?(i.recorrido)$ **then**
38: $i.anterior \leftarrow i.siguiente$ ▷ $O(1)$
39: $i.siguiente \leftarrow NULL$ ▷ $O(1)$
40: **else**
41: $i.siguiente \leftarrow Tope(i.recorrido)$ ▷ $O(1)$
42: $i.anterior \leftarrow i.siguiente \rightarrow padre$ ▷ $O(1)$
43: $Desapilar(i.recorrido)$ ▷ $O(1)$
44: **end if**
45: **end if**

Complejidad: $O(n.L)$ tras usar Buscar, $O(L)$ en cualquier otro caso.
Justificación: Las funciones Tope, Apilar y Desapilar son las del módulo Pila(α) del apunte. Esta operación tiene complejidad lineal en la cantidad de nodos n (entrando en el if de la línea 1) cuando el iterador se encuentra en algún nodo sin tener definido su recorrido, lo cual solo puede ocurrir luego de utilizar la función Buscar (lo cual es indicado por $i.busca$).