DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Recuperatorio Trabajo Práctico 2: Diseño

Primer cuatrimestre - 2016 Algoritmos y Estructuras de Datos II

Grupo 22

Integrante	LU	Correo electrónico
BENZO, Mariano	198/14	marianobenzo@gmail.com
FARIAS, Mauro	821/13	farias.mauro@hotmail.com
GUTTMAN, Martin	686/14	mdg_92@yahoo.com.ar

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria – Pabellón I (Planta Baja)
Intendente Güiraldes 2160 – C1428EGA
Ciudad Autónoma de Buenos Aires – Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1 Tipo es Bool

2 Dato(α)

2.1 Interfaz

se explica con TAD DATO

géneros `dato`

Operaciones

TIPO?(**in** d : `dato`) $\longrightarrow$ res : `tipo`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} tipo?(d)\}$
Descripción: Devuelve el tipo del dato ingresado por parametro.
Complejidad: O(1)

VALORNAT(**in** d : `dato`) $\longrightarrow$ res : `nat`
Pre $\equiv$ {Nat?(d)}
Post $\equiv$ $\{res =_{\text{obs}} valorNat(t)\}$
Descripción: Devuelve valor numerido del dato por parametro.
Complejidad: O(1)

VALORSTRING(**in** d : `dato`) $\longrightarrow$ res : `string`
Pre $\equiv$ {String?(d)}
Post $\equiv$ $\{res =_{\text{obs}} valorString(t)\}$
Descripción: Devuelve valor del dato por parametro.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia.

DATONAT(**in** n : α, **in** $tipoDelDato$: `tipo`) $\longrightarrow$ res : `dato`
Pre $\equiv$ $\{tipoDelDato\}$
Post $\equiv$ $\{res =_{\text{obs}} datoNat(n, tipoDelDato)\}$
Descripción: Crea un dato de valor numerico.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia.

DATOSTR(**in** n : α, **in** $tipoDelDato$: `tipo`) $\longrightarrow$ res : `dato`
Pre $\equiv$ $\{\neg tipoDelDato\}$
Post $\equiv$ $\{res =_{\text{obs}} datoString(n, tipoDelDato)\}$
Descripción: Crea un dato de valor de letras.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia.

MISMOTIPO?(**in** $d1$: `dato`, *in* $d2$: `dato`) $\longrightarrow$ res : `bool`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} mismoTipo?(d1, d2)\}$
Descripción: Informa si los datos pasados por parametro son del mismo tipo de valor.
Complejidad: O(1)

STRING?(**in** d : `dato`) $\longrightarrow$ res : `bool`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} String?(d)\}$
Descripción: Informa si el dato pasado por parametro es de tipo string.
Complejidad: O(1)

NAT?(**in** d : `dato`) $\longrightarrow$ res : `bool`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} Nat?(d)\}$
Descripción: Informa si el dato pasado por parametro es de tipo nat.
Complejidad: O(1)

MIN(**in** cd : `Conj(dato)`) $\longrightarrow$ res : `dato`
Pre $\equiv$ {¬EsVacio?(cd)}
Post $\equiv$ $\{res =_{\text{obs}} min(cd)\}$
Descripción: Retorna el minimo entre los valores del conjunto de datos pasado por parametro.
Complejidad: O($Cardinal(cd)$)
Aliasing: Retorna res por referencia.

MAX(**in** cd : `Conj(dato)`) $\longrightarrow$ res : `dato`
Pre $\equiv$ {¬EsVacio?(cd)}
Post $\equiv$ $\{res =_{\text{obs}} max(cd)\}$
Descripción: Retorna el maximo entre los valores del conjunto de datos pasado por parametro.
Complejidad: O($Cardinal(cd)$)
Aliasing: Retorna res por referencia.

$\leq$(**in** *d1* : `dato`, *in d2* : `dato`) $\longrightarrow$ res : `bool`
Pre $\equiv$ {mismoTipo?(d1,d2)}
Post $\equiv$ $\{res =_{\text{obs}} d1 \leq d2)\}$
Descripción: Retorna si d1 es menor o igual a d2
Complejidad: O($long(k)$)

2.2 Representación

se representa con `dattupla`⟨Valor : α,
TipoValor : `bool`⟩

Invariante de representación

1. La estructura siempre representa un dato valido en tanto αsea Nat o String

Rep : $\widehat{\texttt{dat}} \longrightarrow$ *boolean*

$(\forall d : \widehat{\texttt{dat}})$
Rep(d) $\equiv$true

Funcion de Abstracción

Abs : dato d $\longrightarrow$ dato/c $\{Rep(d)\}$

Abs(d) $\equiv$ **if** $Nat?(d)$ **then**
$Nat?(c) \wedge valorNat(c) = valorNat(d)$
else
$String?(c) \wedge valorString(c) = valorString(d)$
fi

2.3 Algoritmos

TIPO?(**in** a : `dato`) $\longrightarrow$ res : `bool`
$res \leftarrow a.TipoValor$ O(1)

O(1)

VALORNAT(**in** a : `dato`) $\longrightarrow$ res : `nat`
$res \leftarrow a.Valor$ O(1)

O(1)

VALORSTR(**in** a : `dato`) $\longrightarrow$ res : `string`
$res \leftarrow a.Valor$ O(1)

O(1)

MISMOTIPO?(**in** *d1* : `dato`, *in d2* : `dato`) $\longrightarrow$ res : `bool`
$res \leftarrow tipo?(d1) = tipo?(d2)$ O(1)

O(1)

NAT?(**in** a : `dato`) $\longrightarrow$ res : `bool`
$res \leftarrow tipo?(a)$ O(1)

O(1)

STRING?(**in** a : `dato`) $\longrightarrow$ res : `bool`
$res \leftarrow \neg Nat?(a)$ O(1)

O(1)

MIN(**in** *cd* : `Conj(dato)`) $\longrightarrow$ res : `dato`
itcd $\leftarrow$ CrearItConj(cd)
minimo $\leftarrow$ Siguiente(itcd)

while HaySiguiente(itcd) **do**

if Siguiente(itcd)$\leq$minimo **then**
minimo $\leftarrow$ Siguiente(itcd);

end if
Avanzar(itcr);

end while

O(Cardinal(cd))

MAX(**in** *cd* : `Conj(dato)`) $\longrightarrow$ *res* : `dato`

```
itcd ← CrearItConj(cd)
maximo ← Siguiente(itcd)

while HaySiguiente(itcd) do

    if maximo≤Siguiente(itcd) then
        minimo ← Siguiente(itcd);

    end if
    Avanzar(itcr);

end while
```

O(Cardinal(cd))

≤(**in** *d1* : `dato`, *in d2* : `dato`) $\longrightarrow$ *res* : `bool`

```
if String?(d1) then
    res ← valorStr(d1) ≤ valorStr(d2)

else
    res ← valorNat(d1) ≤ valorNat(d2)

end if
```

O(valorStr(d1))

Algoritmos de operaciones auxiliares

≤(**in** *s1* : `string`, *in s2* : `string`) $\longrightarrow$ *res* : `bool`

```
Natn ← Longitud(d1)
Natm ← Longitud(d2)
Nati ← 0
Natj ← 0
while (i ≤ n) ∧ (j ≤ m) do
    if ord(n[i]) > ord(m[j])) then
        i ← n
    end if
    if ord(n[i]) < ord(m[j])) then
        j ← m
    end if
    i ← i + 1
end while
res ← i ≤ n
```

O(valorStr(d1))

3 Modulo Campo es String

4 Modulo Registro

4.1 Interfaz

se explica con REGISTRO

géneros `reg`

Operaciones

Cuando se utiliza conj se está utilizando el conjunto lineal provisto por la cátedra. En las complejidades C denota el campo de mayor longitud de un conjunto o registro al que acompaña.

NREG() $\longrightarrow$ *res* : `reg`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ $\emptyset$}
Descripción: Crea un registro nuevo,vacio
Complejidad: O(1)

DEFINIDO?(**in** r : `reg in c:campo`) $\longrightarrow$ *res* : `Bool`
Pre $\equiv$ {$true$}
Post $\equiv$ {$res =_{\text{obs}} def?(c, r)$}
Descripción: Indica si el campo esta definido
Complejidad: $O(Longitud(c))$

DEFINIR(**in/out** r : `reg in c:campo`, *in* d : `dato`)
Pre $\equiv$ {$r = r_0$}
Post $\equiv$ {$d =_{\text{obs}} Definir(c, d, r_0)$}
Descripción: Se define s en el diccionario
Complejidad: $O(Longitud(c))$
Aliasing: d se define por referencia

BORRAR(**in/out** r : `reg in c:campo`)
Pre $\equiv$ {$r =_{\text{obs}} r_0 \wedge def?(c, r)$}
Post $\equiv$ {$r =_{\text{obs}} Borrar(c, r_0)$}
Descripción: Elimina el campo c
Complejidad: $O(Longitud(c))$

SIGNIFICADO(**in** r : `reg in c:campo`) $\longrightarrow$ *res* : `dato`
Pre $\equiv$ {$def?(c, r)$}
Post $\equiv$ {$res =_{\text{obs}} obtener(c, r)$}
Descripción: Se retorna el significado de c
Complejidad: $O(Longitud(c))$

CAMPOS(**in** r : `reg`) $\longrightarrow$ *res* : `conj(campo)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} alias(res, claves(r))$}
Descripción: Devuelve un conjunto de campos que son claves del registro ingresado por parametro
Complejidad: O(1)
Aliasing: Se devuelve el conjunto por referencia, hay Aliasing

BORRAR?(**in** *crit* : `reg`, *in* r : `reg`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {#campos(crit) = 1}
Post $\equiv$ $\{res =_{\text{obs}} borrar?(crit, r)\}$
Descripción: Devuelve true si y solo si todos los campos de crit pertenecen a campos de r.
Complejidad: O($Longitud(dameUno(campos(crit)))$)

COINCIDEALGUNO(**in** r_1 : `reg`, *in* cc : `conj(campo)`, *in* r_2 : `reg`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {cc $\subseteq$ $campos(r_1)\cap$ campos(r_2)}
Post $\equiv$ $\{res =_{\text{obs}} coincideAlguno(r_1, r_2)\}$
Descripción: Devuelve true si y solo si alguno de los campos(dato) de cc pertenece a r1 y r2
Complejidad: O($\#cc * C$)

COINCIDENTODOS(**in** r_1 : `reg`, *in* cc : `conj(campo)`, *in* r_2 : `reg`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {cc $\subseteq$ $campos(r_1)\cap$ campos(r_2)}
Post $\equiv$ $\{res =_{\text{obs}} coincidenTodos(r_1, cc, r_2)\}$
Descripción: Devuelve ture si y solo si todos los campos(dato) de cc pertenecen a r1 y r2
Complejidad: O($\#cc * C$)

ENTODOS(**in** c : `campo`, *in* cr : `conj(registro)`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} enTodos(c, cr)\}$
Descripción: Devuelve true si y solo si campo c pertenece a los campos de cada uno de los registros cr
Complejidad: O($\#(cr)$)

UNIRREGISTROS(**in** c : `campo`, *in* r_1 : `reg`, *in* r_2 : `reg`) $\longrightarrow$ *res* : `registro`
Pre $\equiv$ $\{c \in campos(r_1) \wedge c \in (campos(r_2))\}$
Post $\equiv$ $\{res =_{\text{obs}} combinarTodos(c, r_1, ag(\emptyset, r_2)))\}$
Descripción: Devuelve el registro que combina los valores de r_1 y r_2
Complejidad: O($\#campos(r_1) * C$)
Aliasing: r_1 y r_2 son tomados por referencia

COMBINARTODOS(**in** c : `campo`, *in* r : `reg`, *in* cr : `conj(reg)`) $\longrightarrow$ *res* : `registro`
Pre $\equiv$ $\{c \in campos(r_1) \wedge enTodos(c, cr))\}$
Post $\equiv$ $\{res =_{\text{obs}} combinarTodos(c, r_1, cr)\}$
Descripción: Devuelve el registro que combina los valores de r_1 y un registro apropiado de cr
Complejidad: O($\#campos(r_1) * C * \#(cr)$)
Aliasing: r_1 y r_2 son tomados por referencia

4.2 Representación

se representa con `dlex`
donde `dlex` **es** `subdic: diccString(campo,dato)`

Invariante de representación

Rep : $\widehat{\texttt{Dicc}} \longrightarrow$ *boolean*

($\forall d : \widehat{\texttt{Dicc}}$)
Rep(d) $\equiv$true

Función de abstracción

$\text{Abs} : \widehat{\texttt{reg}}\ d \longrightarrow \widehat{\texttt{reg}}$ $\{\text{Rep}(d)\}$

$(\forall d : \widehat{\texttt{reg}})$
$\text{Abs}(d) \equiv r : \widehat{\texttt{reg}}\ |\text{Abs(dlex)}$

4.3 Algoritmos

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

$Definir(r, c, d)$	O(Longitud(c))
	O(Longitud(c))

iDefinido?(**in** r : `reg`, *in* c : `campo`) $\longrightarrow$ *res* : `bool`

$res \leftarrow Definido?(r, c)$	O(Longitud(c))
	O(Longitud(c))

iSignificado(**in** r : `reg`, *in* c : `campo`) $\longrightarrow$ *res* : `dato`

$res \leftarrow significado(c, r.subdic)$	O(Longitud(c))
	O(Longitud(c))

iBorrar(**in/out** r : `reg`, *in* c : `campo`)

$Borrar(r.subdic)$	O(Longitud(c))
	O(Longitud(c))

campos(**in** r : `reg`) $\longrightarrow$ *res* : `Conj(campo)`

res $\leftarrow$ DiccClaves(r.subdic)	O(1)
	O(1)

borrar?(**in** *crit* : `reg in r: reg`) $\longrightarrow$ *res* : `bool`

res $\leftarrow$ coincidenTodos(crit, campos(crit), r):	O(Longitud(dameUno(campos(crit))))
	O(Longitud(dameUno(campos(crit))))

iCoincideAlguno(**in** *r1* : `reg in cc: conj(campo) in r2: reg`) $\longrightarrow$ *res* : `bool`

$it \leftarrow CrearIt(cc)$	O(1)
$res \leftarrow false$	O(1)
while $(\not res) \wedge HaySiguiente(it)$ **do**	O(#*cc*)
$res \leftarrow Significado(r1, Siguiente(it)) = Significado(r2, Siguiente(it))$	O($longitud(siguiente(it))$)=O(C)
end while	
	O($\#cc * C$)

iCoincidenTodos(**in** *r1* : `reg in cc: conj(campo) in r2: reg`) $\longrightarrow$ *res* : `bool`

$it \leftarrow CrearIt(cc)$	O(1)
$res \leftarrow true$	O(1)
while $res \wedge HaySiguiente(it)$ **do**	O(#*cc*)
$res \leftarrow Significado(r1, Siguiente(it)) = Significado(r2, Siguiente(it))$	O(*longitud*(*siguiente*(*it*)))=O(C)
end while	
	O(#*cc* ∗ *C*)

Donde C es la longitud del campo más largo en cc

iEnTodos(**in** *c* : `campo in cr: conj(reg)`) $\longrightarrow$ *res* : `bool`

res ← True	O(1)
it ← CrearIt(cr)	O(1)
while haySiguiente(it) ∧ res **do**	O(#(cr))
res ← Definido?(c,siguiente(it))	O(1)
Avanzar(it)	
end while	
	O(#(cr))

iCombinarTodos(**in** *c* : `campo in r1: reg in cr: conj(reg)`) $\longrightarrow$ *res* : `reg`

it ← CrearIt(cr)	O(1)
Bool f ← true	
while haySiguiente(it) ∧ f **do**	O(#(*cr*))
if Significado(r,c)=Significado(Siguiente(it),c) **then**	O(1)
res ← unirRegistros(c,r, siguiente(it))	O(#*campos*(r_1) ∗ *C*)
f ← false	O(1)
end if	
Avanzar(it)	
end while	
	O(#*campos*(r_1) ∗ *C* ∗ #(*cr*))

iUnirRegistros(**in** *c* : `campo in` r_1`: reg in` r_2`: reg`) $\longrightarrow$ *res* : `reg`

$res \leftarrow r_2$	
$it \leftarrow CrearIt(campos(r_1))$	
while HaySiguiente(it) **do**	O(#*campos*(r_1))
$Definir(res, Siguiente(it), Significado(r_1, Siguiente(it)))$	O(*longitud*(*siguiente*(*it*)))=O(C)
end while	
	O(#*campos*(r_1) ∗ *C*)

5 Tabla

5.1 Interfaz

se explica con TABLA, DICCSTRING(STRING, ALFHA),

DiccNat(nat, beta),Nat, String, Dato, Campo, Tipo, Registro, ConjString, ConjNat.

géneros `tabla`

Operaciones

NOMBRE(**in** t : `tab`) $\longrightarrow$ res : `string`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} nombre(t)\}$
Descripción: Devuelve el nombre de la tabla ingresada por parametro.
Complejidad: O(1)
Aliasing: Se retorna res por copia, por ser un tipo basico.

CLAVES(**in** t : `tab`) $\longrightarrow$ res : `Conj(campo)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} claves(t)\}$
Descripción: Devuelve un conjunto de campos clave en la tabla ingresada por parametro.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia, no es modificable.

INDICES(**in** t : `tab`) $\longrightarrow$ res : `ConjString(campo)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} indices(t)\}$
Descripción: Devuelve un conjunto campos con los que se crearon los indices.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia, no es modificable.

CAMPOS(**in** t : `tab`) $\longrightarrow$ res : `ConjString(campo)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} campos(t)\}$
Descripción: Devuelve un conjunto de todos los campos de la tabla.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia, no es modificable.

TIPOCAMPO(**in** c : `campo`, **in** t : `tab`) $\longrightarrow$ res : `tipo`
Pre $\equiv$ {c $\in$ campos(t)}
Post $\equiv$ $\{res =_{\text{obs}} tipoCampo(t)\}$
Descripción: Devuelve el tipo del campo c en la tabla.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia, no es modificable.

REGISTROS(**in** t : `tab`) $\longrightarrow$ res : `itConj(registro)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} registros(t)\}$
Descripción: Devuelve un conjunto a los registros de la tabla.
Complejidad: O(1)
Aliasing: Se devuelve res referencia.

CANTIDADDEACCCESOS(**in** t : `tab`) $\longrightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} cantidadDeAccesos(t)\}$
Descripción: Devuelve la cantidad de modificaciones de la tabla ingresada por parametro.
Complejidad: O(1)
Aliasing: Se devuelve res por copia.

NUEVATABLA(**in** *nombre* : `string`, *in claves* : `conjString(campo)`, *in columnas* : `registro`) $\longrightarrow$ res : `tab`
Pre $\equiv$ {$\neg\emptyset$?(claves) $\wedge$ claves$\subseteq$campos(columnas)}
Post $\equiv$ $\{res =_{\text{obs}} nuevaTabla(t)\}$
Descripción: Crea una tabla sin registros.
Complejidad: O(1)

AGREGARREGISTRO(**in** r : `registro`, *in/out* t : `tab`)
Pre $\equiv$ {t_0=t $\wedge$ campos(r)=$_{\text{obs}}$campos(t) $\wedge$ puedorInsertar?(r,t)}
Post $\equiv$ {agregarRegistro(r,t_0)}
Descripción: Agrega un registro a la tabla pasada por parametro.
Complejidad: O($Log(n)$)
Aliasing: Agrega el registro r por referencia.

BORRARREGISTRO(**in** *crit* : `registro`, *in/out* t : `tab`)
Pre $\equiv$ {t_0=t $\wedge$ #(campos(r))=1 $\wedge_{\text{L}}$ Siguiente(CrearIt(campos(crit)))$\in$claves(t)}
Post $\equiv$ {borrarRegistro(r,t_0)}
Descripción: Borra los registros que cumplan el criterio pasado por parametro.
Complejidad: O(n)

INDEXAR(**in** *crit* : `registro`, *in/out* t : `tab`)
Pre $\equiv$ {t_0=t $\wedge$ puedeIndexar(c,t)}
Post $\equiv$ {indexar(c,t_0)}
Descripción: Crea un indice en base al campo de crit.
Complejidad: O(n)

PUEDOINSERTAR?(**in** r : `registro`, *in* t : `tab`) $\longrightarrow$ res : `bool`
Pre $\equiv$ {$true$}
Post $\equiv$ $\{res =_{\text{obs}} puedoInsertar?(r, t)\}$
Descripción: Informa si el registro pasado por parametro no tiene valores repetidos con respectos a los registros existentes, para los campos clave en la tabla pasada por parametro.
Complejidad: O(n)
Aliasing: Retorna res por referencia, no es modificable.

COMPATIBLE(**in** r : `registro`, *in* t : `tab`) $\longrightarrow$ res : `bool`
Pre $\equiv$ {$true$}
Post $\equiv$ $\{res =_{\text{obs}} compatible(r, t)\}$
Descripción: Informa si el registro pasado por parametro tiene correspondecia en los tipos de los campos de tabla pasada por parametro.
Complejidad: O(1)
Aliasing: Retorna res por referencia, no es modificable.

MINIMO(**in** c : `campo`, *in* t : `tab`) $\longrightarrow$ res : `dato`
Pre $\equiv$ {$\neg\emptyset$?(registro(t)) $\wedge$ c$\in$indices(t)}
Post $\equiv$ $\{res =_{\text{obs}} minimo(c, t)\}$
Descripción: Retorna el minimo entre los valores de la tabla para el campo c.
Complejidad: O($L + Log(n)$)
Aliasing: Retorna res por referencia.

MAXIMO(**in** c : `campo`, *in* t : `tab`) $\longrightarrow$ *res* : `dato`
Pre $\equiv$ {$\neg\emptyset$?(registro(t)) $\wedge$ c$\in$indices(t)}
Post $\equiv$ {$res =_{\text{obs}} maximo(c, t)$}
Descripción: Retorna el maximo entre los valores de la tabla para el campo c.
Complejidad: O($L + Log(n)$)
Aliasing: Retorna res por referencia.

PUEDEINDEXAR(**in** c : `campo`, *in* t : `tab`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {$True$}
Post $\equiv$ {$res =_{\text{obs}} puedeIndexar(c, t)$}
Descripción: Informa si se puede crear un nuevo indice.
Complejidad: O(1)
Aliasing: Retorna res por referencia, no es modificable.

COINCIDENCIAS(**in** r : `registro`, *in cj* : `Conj(registro)`) $\longrightarrow$ *res* : `Conj(ItConj(registro))`
Pre $\equiv$ {$True$}
Post $\equiv$ {$res =_{\text{obs}} coincidencias(r, cj)$}
Descripción: Devuelve el conjunto de registros de la tabla, que coinciden con los valores de r.
Complejidad: O($Cardinal(cj)$)
Aliasing: Retorna res por referencia.

HAYCOINCIDENCIA(**in** r : `registro`, **in** *cjc* : `Conj(campo)`, *in cjr* : `Conj(registro)`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {$True$}
Post $\equiv$ {$res =_{\text{obs}} hayCoincidencia(r, cjc, cjr)$}
Descripción: Retorna true si algun registro del conjunto cjr, coincide con r en todos los valores de los campos de cjc.
Complejidad: O($Cardinal(cjr)$)
Aliasing: Retorna res por referencia, no es modificable.

COMBINARREGISTROS(**in** c : `campo`, **in** *cj1* : `Conj(registro)`, *in cj2* : `Conj(registro)`) $\longrightarrow$ *res* : `conj(registro)`
Pre $\equiv$ {$True$}
Post $\equiv$ {$res =_{\text{obs}} combinarRegistros(c, cj1, cj2)$}
Descripción: Combina los valores de los registros para el campo dado por parametro.
Complejidad: O($Cardinal(cj1) + Cardinal(cj2)$)
Aliasing: Retorna res por referencia, es modificable.

DAMECOLUMNA(**in** c : `campo`, *in cj* : `Conj(registro)`) $\longrightarrow$ *res* : `Conj(dato)`
Pre $\equiv$ {$True$}
Post $\equiv$ {$res =_{\text{obs}} dameColumna(c, cj1, cj2)$}
Descripción: Reune en un conjunto los valores del campo pasado por parametro.
Complejidad: O($\#cr * log(\#cr) + (\#cr * long(k))$)
Aliasing: Retorna res por referencia, no es modificable.

MISMOSTIPOS(**in** r : `registro`, **in** t : `tab`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {campos(r)$\subseteq$campos(t)}
Post $\equiv$ {$res =_{\text{obs}} mismosTipos(r, t)$}
Descripción: Compara los tipos correspondientes a los campos del registro y la tabla.
Complejidad: O(1)
Aliasing: Retorna res por referencia, no es modificable.

BuscarEnTabla(**in** c : `campo`, **in** t : `tab`) $\longrightarrow$ res : `Conj(ItConj(registro))`
Pre $\equiv$ {c⊆campos(t)}
Post $\equiv$ $\{res =_{\text{obs}} BuscarEnTabla(c, t)\}$
Descripción:
Complejidad: O($O(L + Log(n))$)
Aliasing: Retorna res por referencia, es modificable.

5.2 Representación

se representa con `Tabla`
donde `tab` **es** `tupla`⟨Nombre : `NombreTabla`,
Registros : `Conj(Registro)`,
Campos : `DiccString(Campo,Tipo)`,
Claves : `Conj(Campo)`,
IndiceS : `tupla`⟨CampoI : `campo`, ,
EnUso : `bool`,
Indice : `DiccString(string, Conj(ItConj(Registro)))`,
Min : `Dato`,
Max : `Dato`⟩
IndiceN : `tupla`⟨CampoI : `campo`, ,
EnUso : `bool`,
Indice : `DiccNat(nat, Conj(ItConj(Registro)))`,
Min : `Dato`,
Max : `Dato`⟩
RelacionInd : `tupla`⟨CampoR : `campo`, ,
ConsultaN : `DiccNat(nat, Acceso))`,
ConsultaS : `DiccString(string, Acceso))`⟩
#Accesos : `Nat`⟩
`donde Acesso es <S: ItConj(ItConj(Registro)),N: ItConj(ItConj(Registro))>`

Invariante de representación

1. t.Claves esta inclido o es igual a t.Campos.
2. t.Nombre es un string acotado.
3. Para todo registro r de t.Registros, entonces Campos(r) es igual al t.Campos.
4. Para todo registro r de t.Registros y para todo campo c de Campos(r), entonces Tipo?(Significado(r,c)) es igual Significado(t.Campos, c).
5. Si t.IndiceS.EnUso es true y t.IndiceS.CampoI pertenece a t.Campos, para todo string d, si Definido?(t.IndiceS.Indice, d) es true, entonces para todo itConj(registro) it que pertenece a
el conj(ItConj(registro)) cj Significado(t.IndiceS.Indice, d), registro r ←Siguiente(it), r pertenece a t.Registros.
6. Si t.IndiceS.EnUso es true y t.IndiceS.CampoI pertenece a t.Campos, entonces para todo registro R de t.Registros entonces Definido?(t.IndiceS.Indice, Significado(r, t.IndiceS.CampoI)) es true, entonces algun ItConj(registro) it que pertenece a Conj(itConj()) cj Significado(t.IndiceS.Indice, Significado(r, t.IndiceS.CampoI)) entonces Siguiente(it)=R.

7. Si t.IndiceN.EnUso es true y t.IndiceN.CampoI pertenece a t.Campos, para todo string d, si Definido?(t.IndiceN.Indice, d) es true, entonces para todo itConj(registro) it que pertenece a
el conj(ItConj(registro)) cj Significado(t.IndiceS.Indice, d), registro r ←Siguiente(it), r pertenece a t.Registros.

8. Si t.IndiceN.EnUso es true y t.IndiceN.CampoI pertenece a t.Campos, entonces para todo registro R de t.Registros entonces Definido?(t.IndiceN.Indice, Significado(r, t.IndiceN.CampoI)) es true, entonces algun ItConj(registro) it que pertenece a Conj(itConj()) cj Significado(t.IndiceN.Indice, Significado(r, t.IndiceN.CampoI)) entonces Siguiente(it)=R.

9. El campo de e.RelacionInd.CampoR pertenece a t.claves

10. Si t.IndiceS.EnUso es true y para todo string X, ¬tipoCampo(t.campo, e.RelacionInd.CampoR) y Definido?(t.RelacionInd.ConsultaS, X), tal que Acceso dataInd ←Significado(t.RelacionInd.ConsultaN, Y), y el registro R ←Siguiente(Siguiente(dataInd.S)), entonces R pertenece a t.registros $\wedge_{\text{L}}$ ValorString(Significado(R, t.RelacionInd.campoR))=Y

11. Si t.IndiceN.EnUso es true y para todo nat Y, ¬tipoCampo(t.campo, e.RelacionInd.CampoR) y Definido?(t.RelacionInd.ConsultaN, Y), tal que el Acceso dataInd ←Significado(t.RelacionInd.ConsultaN, Y), y el registro R ←Siguiente(Siguiente(dataInd.N)), entonces R pertenece a t.registros $\wedge_{\text{L}}$ ValorString(Significado(R, t.RelacionInd.campoR))=Y

12. Si t.IndiceS.EnUso es true y para todo nat Y, tipoCampo(t.campo, e.RelacionInd.CampoR) y Definido?(t.RelacionInd.ConsultaN, Y), tal que el Acceso dataInd ←Significado(t.RelacionInd.ConsultaN, Y), y el registro R ←Siguiente(Siguiente(dataInd.S)), entonces R pertenece a t.registros $\wedge_{\text{L}}$ ValorString(Significado(R, t.RelacionInd.campoR))=Y

13. Si t.IndiceN.EnUso es true y para todo nat Y, tipoCampo(t.campo, e.RelacionInd.CampoR) y Definido?(t.RelacionInd.ConsultaN, Y), tal que el Acceso dataInd ←Significado(t.RelacionInd.ConsultaN, Y), y el registro R ←Siguiente(Siguiente(dataInd.N)), entonces R pertenece a t.registros $\wedge_{\text{L}}$ ValorString(Significado(R, t.RelacionInd.campoR))=Y

14. El valor de e.#Accesos debe ser la cantidad de registros agregados, la cantidad de registros borrados

Función de abstracción

$\text{Abs} : \widehat{\texttt{tab}}\ s \longrightarrow \widehat{\texttt{Tabla}}$ $\{\text{Rep}(s)\}$

$(\forall s : \widehat{\texttt{tab}})$
$\text{Abs}(s) \equiv t : \widehat{\texttt{Tabla}} \mid s.Nombre =_{\text{obs}} nombre(t) \wedge s.Claves =_{\text{obs}} claves(t) \wedge$
$s.Indices =_{\text{obs}} indices(t) \wedge s.Registros =_{\text{obs}} registros(t) \wedge DiccClaves(s.Campos) =_{\text{obs}} campos(t)$
$\wedge\ s.\#Accesos =_{\text{obs}} cantidadDeAccesos(t) \wedge$
$((\forall c : \text{campo}) Definido?(s.Campos, c)\ \Rightarrow_{\text{L}}\ Significado(s.Campos, c) =_{\text{obs}} tipoCampo(c, t))$

5.3 Algoritmos

NOMBRE(**in** t : `tab`) $\longrightarrow$ res : `string`

$res \leftarrow t.Nombre$	O(1) por ref
	O(1) por ref

CLAVES(**in** t : `tab`) $\longrightarrow$ res : `Conj(campo)`

res ← t.Claves	O(1) por ref
	O(1) por ref

INDICES(**in** t : `tab`) $\longrightarrow$ res : `ConjString(campo)`

ConjString(campo) res ← vacio();	O(1) por ref
if t.IndiceS.EnUso **then**	O(1) por ref
AgregarRapido(res, t.IndiceS.CampoI)	O(1) por ref
end if	
if t.IndiceN.EnUso **then**	O(1) por ref
AgregarRapido(res, t.IndiceN.CampoI)	O(1) por ref
end if	
	O(1) por ref

CAMPOS(**in** t : `tab`) $\longrightarrow$ res : `ConjString(campo)`

$res \leftarrow DiccClaves(t.Campos)$	O(1) por ref
	O(1) por ref

TIPOCAMPO(**in** c : `campo`, *in* t : `tab`) $\longrightarrow$ res : `Tipo`

$res \leftarrow Significado(t.Campos, c)$	O(1) por ref
	O(1) por ref

REGISTROS(**in** t : `tab`) $\longrightarrow$ res : `Conj(registro)`

$res \leftarrow t.registros$	O(1) por ref
Se retorna el conjunto por referencia	
	O(1) por ref

CANTDEACCESOS(**in** t : `tab`) $\longrightarrow$ res : `nat`

$res \leftarrow t.cantDeAccesos$	O(1) por ref
	O(1) por ref

NUEVATABLA(**in** *nombre* : `string`, *in claves* : `conj(campo)`, *in columnas* : `registro`) $\longrightarrow$ *res* : `tab`

Conj(registro) Registros $\leftarrow$ Vacio()	O(1) por ref
DiccString(campo, tipo) Campos $\leftarrow$ Vacio()	O(1) por ref
Campo c $\leftarrow$Siguiente(CrearIt(claves))	
Dato d $\leftarrow$ Obtener(columnas, Siguiente(CrearIt(claves)))	
IndiceS $\leftarrow < c, False, Vacio(), d, d >$	O(1) por ref
IndiceN $\leftarrow < c, False, Vacio(), d, d >$	O(1) por ref
#Acessos $\leftarrow$ 0	O(1) por ref
RelacionInd $\leftarrow < c, Vacio(), Vacio() >$	
res $\leftarrow < nombre, Registros, Campos, claves, IndiceS, IndiceN, RelacionInd, 0 >$	O(1) por ref
itcampos $\leftarrow$ crearIt(Campos(columnas))	O(# de campos)
while HaySiguiente(itcampos) **do**	O(# de campos)
Dato valor $\leftarrow$ Significado(columnas, Siguiente(itcampos))	O(L)
Definir(res.Campos, Siguiente(itcampos), Tipo?(valor))	O(1) por ref
Avanzar(itcampos)	O(1) por ref
end while	
Donde L es la longitud del valor string mas largo.	
	O((#(campos(columnas))*L)

```
AGREGARREGISTRO(in r : registro, in/out t : tab)
  Para poder acceder al registro en el conj en O(1) por ref, guardo el iterador al elemento
  itConj(Registro) nuevo ← AgregarRapido(t.Registros,r)                    O(1) por ref
  t.#Accesos++                                                             O(1) por ref
  Este registro debe ser indexado, si algun indice esta en uso.
  if t.IndiceS.EnUso ∧ t.IndiceN.EnUso then
    Dato valorS ← Significado(r, t.IndiceS.CampoI)                         O(L)
    Dato valorN ← Significado(r, t.IndiceN.CampoI)                         O(Log(n))
    Bool DefinidoS ←Definido?(t.IndiceS.Indice, ValorString(valorS))
                                                                           O(1)
    Bool DefinidoN ←Definido?(t.IndiceN.Indice, ValorNat(valorN))
                                                                           O(Log(n))
    if ¬DefinidoS ∧ ¬DefinidoN then                                        O(1)
      Ambos no estan definidos
      cjS ← vacio()                                                        O(1) por ref
      cjN ← vacio()                                                        O(1) por ref
      newS ←AgregarRapido(cjS, nuevo)                                      O(1) por ref
      newN ←AgregarRapido(cjN, nuevo)                                      O(1) por ref
      Definir(t.IndiceS.Indice, ValorString(valorS), cjS)                  O(1) por ref
      Definir(t.IndiceN.Indice, ValorString(valorN), cjN)                  O(Log(n)) por ref
    else
      if DefinidoS ∧ ¬DefinidoN then                                       O(1)
        cjS ← Significado(t.IndiceS.Indice, ValorString(valorS))
                                                                           O(1)
        cjN ← vacio()                                                      O(1)
        newS ←AgregarRapido(cjS, nuevo)                                    O(1)
        newN ←AgregarRapido(cjN, nuevo)                                    O(1)
        Definir(t.IndiceN.Indice, ValorNat(valorN), cjN)                   O(Log(n)) por ref
      else
        if ¬DefinidoS ∧ DefinidoN then                                     O(1)
          cjN ← Significado(t.IndiceN.Indice, ValorNat(valorN))
                                                                           O(Log(n))
          cjS ← vacio()                                                    O(1)
          newS ←AgregarRapido(cjS, nuevo)                                  O(1)
          newN ←AgregarRapido(cjN, nuevo)                                  O(1)
          Definir(t.IndiceS.Indice, ValorString(valorS), cjS)
                                                                           O(1) por ref
        else
          Caso en que esta definido en los dos indices
          cjN ← Significado(t.IndiceN.Indice, ValorNat(valorN))
                                                                           O(Log(n))
          cjS ← Significado(t.IndiceS.Indice, ValorString(valorS))
                                                                           O(1)
          newS ←AgregarRapido(cjS, nuevo)                                  O(1) por ref
          newN ←AgregarRapido(cjN, nuevo)                                  O(Log(n)) por ref
        end if
      end if
    end if
    if t.IndiceS.Min > valorS then                                         O(L)
      t.IndiceS.Min ← valorS                                               O(1) por ref
    end if
    if valorS > t.IndiceS.Max then                                         O(L)
```

```
        t.IndiceS.Max ← valorS                                  O(1) por ref
    end if
    if t.IndiceN.Min > valorN then                              O(1) por ref
        t.IndiceN.Min ← valorN                                  O(1) por ref
    end if
    if valorN > t.IndiceN.Max then                              O(1) por ref
        t.IndiceN.Max ← valorN                                  O(1) por ref
    end if
else
    if t.IndiceS.EnUso ∧ ¬t.IndiceN.EnUso then
        Obtengo de r el valor del campo con el que se creo el indice.
        Dato valor ← Significado(r, t.IndiceS.CampoI)           O(L)
        Bool Def ←Definido?(t.IndiceS.Indice, ValorString(valor))
        if Def then                                             O(1) por ref
            Si esta definido, entonces hay varios registros que cumplen
            agrego el iterador al conjunto de que ya estaban.
            viejo ← Significado(t.IndiceS.Indice, ValorString(valor))
                                                                O(L)
            itConj(registro) newS ←AgregarRapido(viejo, nuevo)
                                                                O(L)
            itConj(registro) newN ←CrearIt(vacio())
        else
            Conj(Registro) viejo ← Vacio()                      O(1) por ref
            itConj(registro) newS ←AgregarRapido(viejo, nuevo)
                                                                O(1) por ref
            itConj(registro) newN ←CrearIt(vacio())
            Definir(t.IndiceS.Indice, ValorString(valor), viejo)  O(L)
            Como ingresamos un nuevo valor, actualizamos el min y max
            if t.IndiceS.Min > valor then                       O(L)
                t.IndiceS.Min ← valor                           O(L)
            end if
            if valor > t.IndiceS.Max then                       O(L)
                t.IndiceS.Max ← valor                           O(L)
            end if
        end if
    else
        if ¬t.IndiceS.EnUso ∧ t.IndiceN.EnUso then
            Obtengo de r el valor del campo con el que se creo el indice.
            Dato valor ← Significado(r, t.IndiceN.CampoI)       O(L)
            Bool Def ← Definido?(t.IndiceN.Indice, ValorNat(valor))
                                                                O(log(n))
            if Def then                                         O(1) por ref
                Si esta definido, entonces hay varios registros que cumplen
                agrego el iterador al conjunto de que ya estaban.
                viejo ← Significado(t.IndiceN.Indice, ValorNat(valor))
                                                                O(L)
                itConj(registro) newN ←AgregarRapido(viejo, nuevo)
                                                                O(L)
                itConj(registro) newS ←CrearIt(vacio())
            else
                Conj(Registro) viejo ← Vacio()                  O(1) por ref
                itConj(registro) newN ←AgregarRapido(viejo, nuevo)
```

```
                                                        O(1) por ref
            itConj(registro) newS ←CrearIt(vacio())
            Definir(t.IndiceN.Indice, ValorNat(valor), viejo)
                                                        O(L)
            Como ingresamos un nuevo valor, actualizamos el min y max
            if t.IndiceN.Min > valor then               O(1) por ref
                t.IndiceN.Min ← valor                   O(1) por ref
            end if
            if valor > t.IndiceN.Max then               O(1) por ref
                t.IndiceN.Max ← valor                   O(1) por ref
            end if
        end if
    end if
end if
end if
Bool BasadoEn ← Significado(t.campos, t.RelacionInd.CampoR)
if BasadoEn then
    Es True entonces es un DiccNat
    Nat elem ←ValorNat(Significado(r, t.RelacionInd.CampoR))
    Definir(t.RelacionInd.ConsultaN, elem, < newS, newN >)
else
    Es False entonces es un DiccString
    String elem ←ValorString(Significado(r, t.RelacionInd.CampoR))
    Definir(t.RelacionInd.ConsultaS, elem, < newS, newN >)
end if
                                                        O(L+Log(n))
```

```
INDEXAR(in c : campo, in/out t : tab)
  if tipoCampo(c,t.campos) then
     t.IndiceN.EnUso ← True
  else
     t.IndiceS.EnUso ← True
  end if
  ItConj(ItConj(registro)) newS ←CrearIt(CrearIt(Vacio()))
  ItConj(ItConj(registro)) newN ←CrearIt(CrearIt(Vacio()))
  < ItConj(ItConj(registro)), ItConj(ItConj(registro)) > dataIndices
  Buscamos en la Relacion de Indices segun su campo.
  Primero necesito saber en base a que tipo de campo fue creado.
  Tipo BasadoEn ← Significado(t.campos, t.RelacionInd.CampoR)
  itConj(registro) cr ← CrearItConj(t.registros)
  while HaySiguiente(cr) do
     if tipoCampo?(c,t) then
        Caso Naturales
        Dato valor ← Significado(Siguiente(cr), c)
        Bool Definido ←Definido?(t.IndiceN.Indice, ValorNat(valor))
        itConj(registro) itr ← Copiar(cr)
        if Definido then
           Significa que para este valor del indice hay mas de un iterador de conj a reg
           regviejos ← Significado(t.IndiceN.Indice, ValorNat(valor))
           itConj(itConj(registro)) newN ← AgregarRapido(regviejos, itr)
        else
           Conj(itConj(registro)) nuevo ← Vacio()
           newN ← AgregarRapido(nuevo, itr)
           Definir(t.IndiceN.Indice, ValorNat(valor), nuevo)
        end if
     else
        Caso Strings
        Dato valor ← Significado(Siguiente(cr), c)
        Bool Definido ←Definido?(t.IndiceS.Indice, ValorString(valor))
        itConj(registro) itr ← cr
        if Definido then
           Significa que para este valor del indice hay mas de un iterador de conj a reg
           regviejos ← Significado(t.IndiceS.Indice, ValorString(valor))
           itConj(itConj(registro)) newS ← AgregarRapido(regviejos, itr)
        else
           Conj(itConj(registro)) nuevo ← Vacio()
           newS ← AgregarRapido(nuevo, itr)
           Definir(t.IndiceS.Indice, ValorString(valor), nuevo)
        end if
     end if
     Ahora actualizo la relacion de indices
     Dato valorR ← Significado(Siguiente(cr), t.RelacionInd.CampoR)
     Bool DefinidoR
     if BasadoEn then
        La relacion de indices esta basada en un campo Natural
        DefinidoR ← Definido?(t.RelacionInd.ConsultaN, ValorNat(valorR))
     else
        La relacion de indices esta basada en un campo String
        DefinidoR ← Definido?(t.RelacionInd.ConsultaS, ValorString(valorR))
```

```
    end if
    if DefinidoR then
        Significa que el otro indice esta en uso.
        if BasadoEn then
            La relacion de indices esta basada en un campo Natural
            dataIndices ← Significado(t.RelacionInd.ConsultaN, ValorNat(valorR))
            if tipoCampo?(c,t.campos) then
                dataIndices.N ← newN
            else
                dataIndices.S ← newS
            end if
        else
            La relacion de indices esta basada en un campo String
            dataIndices ← Significado(t.RelacionInd.ConsultaS, ValorString(valorR))
            if tipoCampo?(c,t) then
                dataIndices.N ← newN
            else
                dataIndices.S ← newS
            end if
        end if
    else
        Significa que el otro indice no esta en uso.
        dataIndices ← < newS, newN >
        if BasadoEn then
            Definir(t.RelacionInd.ConsultaN, ValorNat(valorR), dataIndices)
        else
            Definir(t.RelacionInd.ConsultaS, ValorString(valorR), dataIndices)
        end if
    end if
    Avanzar(cr)
end while                                                    O(1) por ref
```

PUEDOINSERTAR?(**in** r : `registro`, *in* t : `tab`) $\longrightarrow$ *res* : `bool`

```
res ← campatible(r,t) ∧ ¬hayCoincidencia( r, Campo(r), registros(t) )
                                                             O(Cardinal(registros(t)))
```

O(Cardinal(registros(t)))

COMPATIBLE(**in** r : `registro`, *int* t : `tab`) $\longrightarrow$ *res* : `bool`

```
bool valor ← True
if Cardinal(campos(r))=Cardinal(DiccClaves(t.Campos)) then
    itcampos ← CrearItString(DiccClaves(t.Campos))
    while valor ∧ HaySiguiente(itcampos) do                  O(1) por ref
        Campo c ← Siguiente(itcampos)                        O(1) por ref
        valor ← Definido?(r, c)                              O(1) por ref
    end while
else
    valor ← False                                            O(1) por ref
end if
res ← valor ∧_L mismosTipos(r,t)                             O(1) por ref
El costo del While es O(1) por ref ya que la cantidad de campos de la tabla es acotado
```

O(1)

MINIMO(**in** *c* : `campo`, *in t* : `tab`) $\longrightarrow$ *res* : `dato`

Si hay indice en el campo c, debe ser de complejidad O(1) por ref	
if t.IndiceS.EnUso $\wedge$ t.IndiceS.CampoI=c **then**	
Sabemos que hay un indice string para el campo c	
res $\leftarrow$ t.IndiceS.Min	O(Cardinal(t.registros))
else	
if t.IndiceN.EnUso $\wedge$ t.IndiceN.CampoI=c **then**	
Sabemos que hay un indice string para el campo c	
res $\leftarrow$ t.IndiceN.Min	O(Cardinal(t.registros))
end if	
end if	
	O(1) por ref

MAXIMO(**in** *c* : `campo`, *in t* : `tab`) $\longrightarrow$ *res* : `dato`

Si hay indice en el campo c, debe ser de complejidad O(1) por ref	
if t.IndiceS.EnUso $\wedge$ t.IndiceS.CampoI=c **then**	
Sabemos que hay un indice string para el campo c	
res $\leftarrow$ t.IndiceS.Max	O(Cardinal(t.registros))
else	
if t.IndiceN.EnUso $\wedge$ t.IndiceN.CampoI=c **then**	
Sabemos que hay un indice string para el campo c	
res $\leftarrow$ t.IndiceN.Max	O(Cardinal(t.registros))
end if	
end if	
	O(1) por ref

PUEDEINDEXAR(**in** *c* : `campo`, *in t* : `tab`) $\longrightarrow$ *res* : `bool`

if TipoCampo(c, t) **then**	
res $\leftarrow \neg$(t.IndiceN.EnUso)	
else	
res $\leftarrow \neg$(t.IndiceS.EnUso)	
end if	
	O(1) por ref

HAYCOINCIDENCIA(**in** *r* : `registro`, *in cc* : `ConjString(campo)`, *in cr* : `Conj(registro)`) $\longrightarrow$ *res* : `bool`

itcr $\leftarrow$ CrearItConj(cr)	O(1) por ref
res $\leftarrow$ false	O(1) por ref
while HaySiguiente(itcr) **do**	O(Cardinal(cr))
res $\leftarrow$ coincideAlguno(r,cc,Siguiente(itcr)) $\vee$res	O(1) por ref
Avanzar(itcr)	O(1) por ref
end while	
	O(Cardinal(cr))

COINCIDENCIAS(**in** *crit* : `registro`, *in cr* : `Conj(registro)`) $\longrightarrow$ *res* : `Conj(ItConj(registro))`

Conj(registro) salida ← Vacio()	O(1) por ref
Debemos comparar todos los registros de cr.	
y agregarlos al conjunto de registros salida	
itcr ← CrearItConj(cr)	
while HaySiguiente?(cr) **do**	O(Cardinal(cr))
if coincidenTodos(crit,campos(crit),Siguiente(itcr)) **then**	O(1) por ref
AgregarRapido(salida, CrearIt(Siguiente(itcr)))	O(1) por ref
AgregarRapido(salida, itcr)	O(1) por ref
end if	
Avanzar(itcr)	O(1) por ref
end while	
res ← salida	O(1) por ref
	O(Cardinal(cr))

COMBINARREGISTROS(**in** *c* : `campo`, *in cr1* : `Conj(registro)`, *in cr2* : `Conj(registro)`) $\longrightarrow$ *res* : `Conj(registros)`

itcr1 ← CrearItConjString(cr1)	O(1) por ref
Lo malo es que combinarRegistros sera O(Cardinal(cr2)*Log(Cardinal(cr2)))	
if Cardinal(cr2)≥1 **then**	O(1) por ref
Registro rtemp ← Siguiente(CrearIt(cr2))	O(1) por ref
Tipo rac ← Tipo?(Significado(rtemp, c))	O(1) por ref
if rac **then**	O(1) por ref
Caso Natural	
DiccNat(Nat, Conj(registro)) d ← vacio()	O(1) por ref
itcr2 ← CrearIt(cr2)	O(1) por ref
while HaySiguiente?(itcr2) **do**	O(1) por ref
Dato valor ← Obtener(Siguiente(itcr2), c)	O(1) por ref
if Definido?(d, ValorNat(valor)) **then**	O(Cardinal(cr2))
cjViejo ← Significado(d, ValorNat(valor))	O(Cardinal(cr2))
AgregarRapido(d, Siguiente(itcr2))	O(1) por ref
else	
ConjNat(registro) cjNuevo ← vacio()	O(1) por ref
AgregarRapido(d, Siguiente(itcr2))	O(1) por ref
Definir(d, ValorNat(valor), cjNuevo)	O(Cardinal(cr2))
end if	
Avanzar(itcr2)	O(1) por ref
end while	
else	
Caso String	
DiccString(String, Conj(registro)) d ← vacio()	O(1) por ref
itcr2 ← CrearIt(cr2)	O(1) por ref
while HaySiguiente?(itcr2) **do**	O(1) por ref
Dato valor ← Obtener(Siguiente(itcr2), c)	O(1) por ref
if Definido?(d, ValorString(valor)) **then**	O(Cardinal(cr2))
cjViejo ← Significado(d, ValorString(valor))	O(Cardinal(cr2))
AgregarRapido(d, Siguiente(itcr2))	O(1) por ref
else	
ConjString(registro) cjNuevo ← vacio()	O(1) por ref
AgregarRapido(d, Siguiente(itcr2))	O(1) por ref
Definir(d, ValorString(valor), cjNuevo)	O(Cardinal(cr2))
end if	

```
            Avanzar(itcr2)                                          O(1) por ref
        end while
    end if
    itcr1 ← CrearIt(cr1)                                            O(1) por ref
    res ← vacio()
    while HaySiguiente(itcr1) do                                    O(Cardinal(cr1))
        AgregarRapido(res, combinarTodos(c,Siguiente(itcr1),cr2))
                                                                    O(Cardinal(cr2))
        Avanzar(itcr1)                                              O(1) por ref
    end while
else
    res ← vacio()
end if
                                                                    ---------------
                                                                    O(Cardinal(cr1))
```

DAMECOLUMNA(**in** c : `campo`, *in* cr : `Conj(registro)`) $\longrightarrow$ *res* : `Conj(dato)`

```
res ← vacio()                                                       O(1)
it ← CrearIt(cr)                                                    O(1)
Tvalor ← Tipo?(Significado(Siguiente(it), c))                       O(1)
if Tvalor then
    DiccNat(nat,dato) bolsaN ← vacio                                O(1)
    while haySiguiente(it) do                                       O(#cr)
        Definir(bolsaN, valorNat(Significado(Siguiente(it),c)), Significado(Siguiente(it),c))
                                                                    O(log(#cr))
        avanzar(it)                                                 O(1)
    end while
    itN ← diccClaves(bolsaN)                                        O(1) por ref
    while haySiguiente(itN) do                                      O(#cr)
        AgregarRapido(res, Significado(bolsaN, siguiente(itN)))
                                                                    O(log(#cr))
        avanzar(itN)
    end while
else
    DiccString(String,dato) bolsaS ← vacio()                        O(1)
    while haySiguiente(it) do                                       O(#cr)
        Definir(bolsaS, valorString(Significado(Siguiente(it),c)), Significado(Siguiente(it),c))
                                                                    O(long(k))
        avanzar(it)                                                 O(1)
    end while
    itS ← CrearIt(diccClaves(bolsaS))                               O(1) por ref
    while haySiguiente(itS) do                                      O(#cr)
        AgregarRapido(res, Significado(bolsaS, siguiente(itS)))
                                                                    O(long(siguiente(itS)))
        avanzar(itS)                                                O(1)
    end while
end if
                                                                    ---------------
                                                                    O(#cr * log(#cr) + (#cr * long(k)))
```

MISMOSTIPOS(**in** r : `registro`, *in* t : `tab`) $\longrightarrow$ *res* : `bool`

valor ← True	O(1) por ref
itconjClaves ← CrearIt(Campo(r))	O(1) por ref
while valor $\wedge_L$ HaySiguiente?(itconjClaves) **do**	O(1) por ref
val1← tipo?(Significado(r, Siguiente(itconjClaves)))	O(1) por ref
val2← tipoCampo(Siguiente(itconjClaves), t)	O(1) por ref
valor ← (val1 = val2)	O(1) por ref
Avanzar(cr);	O(1) por ref
end while	
res ← valor	
	O(1) por ref

5.4 Algoritmos operaciones auxiliares

BUSCARENTABLA(**in** *criterio* : `registro`, *in* t : `tab`) $\longrightarrow$ *res* : `Conj(ItConj(registro))`

```
Primero busco que campos de r, estan en los campos de t y son del mismo tipo
itcampos ← DiccClaves(t.campos)
Bool Encontrado ← false
Campo EncontradoCampoInd
Conj(Campo) cj ← vacio()
while HaySiguiente?(itcampos) ∧ ¬Encontrado do
    Campo c ← Siguiente(itcampos)
    Bool Def ←Definido?(criterio, c)
    if Def then
        bool valorD ← (Tipo?(Significado(criterio, c))=Significado(t.campos, c))
        if valorD then
            El campo esta en ambos y es del mismo tipo, lo agrego al conj lineal
            AgregarRapido(cj, c)
            Vemos tambien si el campo c de criterio esta en los indices
            Bool EncS ← (t.IndiceS.EnUso ∧_L t.IndiceS.CampoI=c)
            Bool EncN ← (t.IndiceN.EnUso ∧_L t.IndiceN.CampoI=c)
            Encontrado ← (EncS ∨ EncN)
            if Encontrado then
                EncontradoCampoInd ← c
            end if
        end if
    end if
    Avanzar(itcri)
end while
Si Encontrado es true entonces uso indice, sino recorro todo los registros
if Encontrado then
    Entonces hay un indice para EncontradoCampoInd
    if t.IndiceN.EnUso ∧_L t.IndiceN.CampoI=EncontradoCampoInd then
        Caso Natural
        Obtengo el valor nat del registro criterio
        Nat valor ← ValorNat(Significado(criterio, EncontradoCampoInd))
        Conj(itConj(registro)) res ← Significado(t.IndiceN.Indice, valor)
    end if
    if ¬Significado(t.campos, EncontradoC) then
        Caso String
        Obtengo el valor nat del registro criterio
```

String valor ← ValorString(Significado(criterio, EncontradoCampoInd))
Conj(itConj(registro)) res ← Significado(t.IndiceS.Indice, valor)
end if
elseres ← Coincidencias(criterio, t.registros)
end if
La complejidad depende de si hay indice para algun campo de criterio,
en ese caso la complejidad es O(#(Campos(t))+L + Log(n))
En caso contrario es O(#(Campos(t))+#(Registros(t)))

O(1) por ref

6 NombreTabla es String

7 Base de Datos

7.1 Interfaz

se explica con BASEDEDATOS, NAT, STRING,

tabla, registro, campo, dato, DiccString(String, alfha), DiccNat(Nat, beta)

géneros `base`

Operaciones

TABLAS(**in** b : `base`) $\longrightarrow$ *res* : `ItConjString(NombreTabla)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} nombre(t)\}$
Descripción: Devuelve el nombre de la tabla ingresada por parametro.
Complejidad: O(1)
Aliasing: Se retorna res por copia, por ser un tipo basico.

DAMETABLA(**in** t : `string`, *in* b : `base`) $\longrightarrow$ *res* : `tabla`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} dameTabla(t, b)\}$
Descripción: Devuelve la tabla correspondiente al nombre ingresado por parametro.
Complejidad: O(1)
Aliasing: Se devuelve la tabla correspondiente por referencia.

HAYJOIN?(**in** *t1* : `string`, *in t2* : `string`, *in* t : `base`) $\longrightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} indices(t)\}$
Descripción: Devuelve un conjunto de los indices de la tabla ingresada por parametro.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia y no es modificable.

CAMPOJOIN(**in** *t1* : `string`, *in t2* : `string`, *in* t : `base`) $\longrightarrow$ *res* : `itConjTrie(campo)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} campos(t)\}$
Descripción: Devuelve un conjunto a los campos de la tabla ingresada por parametro.
Complejidad: O(1)
Aliasing: Se devuelve res por referencia.

nuevaDB() $\longrightarrow$ *res* : `base`
Pre $\equiv \{True\}$
Post $\equiv \{res =_{\text{obs}} nuevaDB()\}$
Descripción: Crea una base sin tablas.
Complejidad: O(1)

agregarTabla(**in** *t* : `tabla`, *in b* : `base`)
Pre $\equiv$ {*b*_0=b $\wedge$ nombre(t)$\notin$ tablas(b) $\wedge$ Vacio?(t.registros)}
Post $\equiv$ {agregarTabla(t *b*_0)}
Descripción: Agrega una tabla a la base de datos.
Complejidad: O(1)
Aliasing: Agrega tabla por referencia.

insertarEntrada(**in** *reg* : `registro`, *in t* : `string`, *in b* : `base`)
Pre $\equiv$ {*b*_0=b $\wedge$ t$\in$tablas(b) $\wedge_{\text{L}}$ puedoInsertar?(dameTabla(t) reg)}
Post $\equiv$ {insertarEntrada(r t *b*_0)}
Descripción: Inserta el registro a la tabla t.
Complejidad: O(T)

borrar(**in** *cr* : `registro`, *in t* : `string`, *in b* : `base`)
Pre $\equiv$ {*b*_0=b $\wedge$ t$\in$tablas(b) $\wedge$ #(DiccClaves(cr))=1}
Post $\equiv$ {borrar(cr t *b*_0)}
Descripción: Borra los registros que cumplan el criterio cr pasado por parametro.
Complejidad: O($T + Log(n)$)

generarVistaJoin(**in** *t1* : `string`, *in t2* : `string`, *in c* : `campo`, *in b* : `base`)
Pre $\equiv$ {b_0=b $\wedge$ t1$\sqsupset$t2 $\wedge$ {t1, t2}$\subseteq$tablas(b) $\wedge_{\text{L}}$
Pertenece?(Campos(dameTabla(t1, b)), c) $\wedge$ Pertenece?(Campos(dameTabla(t1, b)), c) $\wedge$
$\neg$hayJoin?(t1, t2, b)}
Post $\equiv$ {generarVistaJoin(cr, t, *b*_0)}
Descripción: Genera el Join, entre las tablas pasadas por parametro.
Complejidad: O($(n + m) * (L + Log((n + m)))$)

borrarJoin(**in** *t1* : `string`, *in t2* : `string`, *in b* : `base`)
Pre $\equiv$ {*b*_0=b $\wedge$ hayJoin?(t1 t2 b)}
Post $\equiv$ {borrarJoin(t1 t2 *b*_0)}
Descripción: Borra el join entre tablas, pasadas por parametro.
Complejidad: O(1)

registros(**in** *t* : `string`, *in b* : `base`) $\longrightarrow$ *res* : `conj(registro)`
Pre $\equiv$ {t$\in$tablas(b)}
Post $\equiv \{res =_{\text{obs}} registros(t\, b)\}$
Descripción: Retorna el conjunto de registros correspondientes al nombre de tabla pasado por parametro
Complejidad: O(1) por ref
Aliasing: Se retorna el conjunto de registros por referencia.

vistaJoin(**in** *t1* : `string`, *in t2* : `string`, *in b* : `base`) $\longrightarrow$ *res* : `conj(registro)`
Pre $\equiv$ {{t1 t2}$\subseteq$tablas(b) $\wedge$ hayJoin?(t1 t2 b)}
Post $\equiv \{res =_{\text{obs}} vistaJoin(t1\, t2\, b)\}$
Descripción: Retorna el conjunto de registros correspondientes al nombre de tabla pasado por parametro
Complejidad: O($R * (L + Log(n * m))$)
Aliasing: Se retorna el conjunto de registros por referencia.

CANTIDADDEACCESOS(**in** t : `string`, *in* b : `base`) $\longrightarrow$ *res* : `nat`
Pre $\equiv$ {t$\in$tablas(b)}
Post $\equiv$ $\{res =_{\text{obs}} cantidadDeAccesos(t\,b)\}$
Descripción: Retorna la cantidad de modificaciones correspondientes al nombre de tabla pasado por parametro.
Complejidad: O(1) por ref
Aliasing: Se retorna res por referencia.

TABLAMAXIMA(**in** b : `base`) $\longrightarrow$ *res* : `string`
Pre $\equiv$ {$\neg\emptyset$?(tablas(b))}
Post $\equiv$ $\{res =_{\text{obs}} tablaMaxima(t\,b)\}$
Descripción: Retorna el nombre de la tabla con la mayor cantidad de modificaciones.
Complejidad: O(1) por ref
Aliasing: Se retorna el nombre de la tabla por referencia, no es modificable.

ENCONTRARMAXIMO(**in** t : `string`, *in ct* : `conj(string)`, *in* b : `base`) $\longrightarrow$ *res* : `string`
Pre $\equiv$ {{t}$\cup$ct $\subseteq$tablas(b)}
Post $\equiv$ $\{res =_{\text{obs}} tablaMaxima(t\,b)\}$
Descripción: Retorna el nombre de alguna de las tablas con mayor cantidad de accesos.
Complejidad: O(1)
Aliasing: Se retorna el nombre de la tabla por referencia, no es modificable.

BUSCAR(**in** *criterio* : `registro`, *in* t : `string`, *in* b : `base`) $\longrightarrow$ *res* : `conj(ItConj(registro))`
Pre $\equiv$ {t$\in$tablas(b)}
Post $\equiv$ $\{res =_{\text{obs}} tablaMaxima(t\,b)\}$
Descripción: Retorna el conjunto de los iteradores de registro al conjunto de registros de la tabla.
Complejidad: O($Cantidadderegistros$)
Aliasing: Se retorna por referencia, es modificable.

7.2 Representación

se representa con `Base`
donde `estr` **es** `tupla`⟨TablaMaxima : `Tmax`,
Tablas : `DiccTrie(NombreTabla; info_tabla)`⟩
donde `info_tabla` **es** `tupla`⟨TActual : `tabla`,
Joins : `DiccTrie(NombreTabla; info_join)`⟩
donde `info_join` **es** `tupla`⟨Rcambios : `Cola(DatoCambio)`,
campoJ : `campo`,
campoT : `tipo`,
JoinS : `DiccTrie(string; itConj(registro))`,
JoinN : `DiccNat(nat; itConj(registro))`,
JoinC : `Conj(registro)`⟩
donde `Tmax` **es** `tupla`⟨NomTabla : `NombreTabla`,
#Modif : `Nat`⟩
donde `DatoCambio` **es** `tupla`⟨Reg : `Registro`,
NomOrigen : `NombreTabla`,
Accion : `Bool`⟩

Invariante de representación

1. El Nombre de las tablas es un String acotado.
2. Índices es un arreglo de tamaño 2, que aloja el Índice correspondiente según el orden de creación.
3. Para toda Dato que es clave en Índice, su significado llamémoslo sign está incluido en Registros.
4. El nombre de la TablaMaxima está definido en Tablas.
5. El nombre de la TablaMaxima corresponde a una del conjunto de tablas más modificadas.
6. En Tmax #Modif es igual a la cantidad de modificaciones de la tabla NomTabla.
7. #Modif de TablaMaxima es mayor o igual a todas las cantidades de modificaciones de las tablas en el conjunto de claves de Tablas.
8. Cada NombreTabla definido en Tablas es igual al nombre de TActual en info_tabla de su significado.
9. Todas las claves de Joins, de todos los significados de las tablas definidas en Tablas, pertenecen al conjunto de claves de Tablas.
10. Para todo info_tabla, el nombre de TActual no está definido en Joins.
11. para todo info_tabla, el nombre de TActual pertenece al conjunto de claves del Joins de las claves de su Joins.
12. Para todo info_join, el tipo de campoJ es igual a campoT.
13. Para todo DatoCambio, NomOrigen es igual a TActual o a la clave que le corresponde en Joins.
14. Si campoT es true JoinS esta vacio, y si es false JoinN esta vacio.
15. Para toda string t1, t2, si Pertenece?(Tablas(b), t1) y Pertenece?(Tablas(b), t1) entonces si HayJoin?(t1,t2, b) y luego si infojoin ← Significado(Significado(b.Tablas, t1).Joins, t2). Si infojoin.CampoT es true entonces para todo nat n, si Definido?(infoJoin.JoinN, n) entonces el registro r ←Siguiente(Significado(infoJoin.JoinN, n)) es tal que ValorNat(Significado(r, infojoin.CampoJ))=n.
16. Para toda string t1, t2, si Pertenece?(Tablas(b), t1) y Pertenece?(Tablas(b), t1) entonces si HayJoin?(t1,t2, b) y luego si infojoin ← Significado(Significado(b.Tablas, t1).Joins, t2). Si infojoin.CampoT es false entonces para todo String s, si Definido?(infoJoin.JoinS, s) entonces
    el registro r ←Siguiente(Significado(infoJoin.JoinS, s)) es tal que ValorNat(Significado(r, infojoin.CampoJ))=s.
17. En DatoCambio, si Accion es true entonces Reg pertence al conjunto de registros de NomOrigen.
18. En DatoCambio, si Accion es false entonces Reg no pertence al conjunto de registros de NomOrigen.

Función de abstracción

$\text{Abs} : \widehat{\texttt{Base}}\ b \longrightarrow \widehat{\texttt{BaseDeDatos}}$ $\{\text{Rep}(b)\}$

$(\forall b : \widehat{\texttt{Base}})$
$\text{Abs}(b) \equiv cb : \widehat{\texttt{BaseDeDatos}}\ |$

$Tablas(b) =_{\text{obs}} tablas(cb) \wedge$
$((\forall n : \text{NombreTabla}) Pertenece?(n, Tablas(b)) \Rightarrow_{\text{L}} DameTabla(n, b) =_{\text{obs}} dameTabla(n, cb))$
$((\forall n1, n2 : \text{NombreTabla})(Definido?(n1, b.tablas) \wedge Definido?(n2, b.tablas)) \Rightarrow_{\text{L}} HayJoin?(n, b) =_{\text{obs}}$
$HayJoin?(n, cs) \wedge_{\text{L}} campoJoin(n1, n2, b) = campoJoin(n1, n2, cb))$

7.3 Algoritmos

TABLAS(**in** b : `estr`) $\longrightarrow$ res : `ConjTrie(string)`

$res \leftarrow DiccClaves(b.tablas)$	O(1) por ref
	O(1) por ref

DAMETABLA(**in** t : `string`, *in* b : `estr`) $\longrightarrow$ res : `tabla`

$info \leftarrow Significado(b.tablas, t)$	O(1) por ref
$res \leftarrow info.TActual$	O(1) por ref
	O(1) por ref

HAYJOIN?(**in** *t1* : `string`, *in t2* : `string`, *in* b : `estr`) $\longrightarrow$ res : `bool`

res ←(Definido?(Obtener(b, t1).Joins,t2) ← Definido?(Obtener(b, t2).Joins,t1))	O(1) por ref
	O(1) por ref

CAMPOJOIN(**in** *t1* : `string`, *in t2* : `string`, *in* b : `estr`) $\longrightarrow$ res : `campo`

res← Obtener(Obtener(b, t1).Joins, t2).campoJ	O(1) por ref
	O(1) por ref

NUEVADB() $\longrightarrow$ res : `estr)`

String s	O(1)
Nat n ← 0	O(1)
TablaMaxima ← $< s, 0 >$	O(1) por ref
res ← $\langle TablaMaxima, vacio() \rangle$	O(1) por ref
	O(1) por ref

AGREGARTABLA(**in** t : `tabla`, *in/out* b : `estr`)

info_tabla ← $\langle cantidadDeAccesos(t), t, vacio() \rangle$	O(1) por ref
Definir(b.tablas, nombre(t), info_tabla)	O(1) por ref
	O(1) por ref

INSERTARENTRADA(**in** *reg* : `registro`, *in t* : `string`, *in/out b* : `estr`)

Obtenemos la tabla es O(1) por ref porque su nombre esta acotado.
info_tabla infoT ← Obtener(b.tablas, t).TActual — O(1) por referencia
Agrego el registro a la tabla. — O(1) por ref
Tabla T ← infoT.TActual — O(1) por ref
agregarRegistro(reg, T) — O(L+Log(n))
Ahora si hay Joins actualizo la informacion temporal de cada Join.
if ¬Vacio?(infoT.Joins) **then** — O(1) por ref
  ItConjString(String) itNomTab ← CrearIt(Claves(infoT.Joins)) — O(1) por ref
  while HaySiguiente?(itNomTab) **do** — O(W)
    info_join infoJ ← Obtener(infoT.Joins, Siguiente(NomTab)) — O(1) por ref
    Encolar(infoJ.Rcambios, $\langle reg, Siguiente(NomTab), true \rangle$) — O(1) por ref
    Encolar es O(L) porque se copia un registro con su cantidad de campos acotada
    y los valores string copiarlos tiene costo O(L), siendo L el valor string mas largo.
    Avanzar(itClaves) — O(1) por ref
  end while
end if
if CantidadDeAccesos(T) b.TablaMaxima.#Modif **then**
  b.TablaMaxima.NomTabla ← Copiar(Nombre(T)) — O(L)
  b.TablaMaxima.#Modif ← Copiar(CantidadDeAccesos(T)) — O(1) por ref
end if
W la cant de tablas en la base

O(W)

BORRAR(**in** *cr* : `registro`, *in t* : `string`, *in/out b* : `estr`)

info_tabla infoT ← Obtener(b.tablas, t).TActual — O(1) por ref
Tabla T ← infoT.TActual — O(1) por ref
NombreTabla NomTab ← NombreTabla(T)
La eliminacion en primera etapa depende de si hay
joins con la tabla pasada por parametro
if ¬∅?(Claves(infoT.Joins)) **then** — O(1) por ref
  Creo el iterador, para navegar los nombres de tablas con los que tiene Join
  itNom ← CrearIt(Claves(infoT.Joins)) — O(1) por ref
  while HaySiguiente?(itNom) **do** — O(Cant de tablas)
    info_join infoJ ← Siguiente(itNom) — O(1) por ref
    Verifico si el Join esta creado en base al
    campo del cr pasado por parametro.
    if infoJ.campoJ=DameUno(Campos(cr)) **then** — O(1) por ref
      Entonces solo actualizo la cola temporal del join
      Registro reg ← Buscar(cr, T) — O(L + Log(n))
      Encolar(infoJ.Rcambios, $\langle reg, Siguiente(itNom), False \rangle$) — O(L)
    else
      Si el campo del criterio de borrado, es distinto que el campo del Join.
      Primero busco que registros coinciden con el criterio
      en el peor caso con complejidad O(cantidad de registros de t) sin indices.
      Conj(Registro) cjc ← Buscar(cr, T) — O(L + Log(n))
      ItConj(Registro) itReg ← CrearIt(cjc)

```
            Luego agrego estos registros a la cola temporal de cambios del join
            while HaySiguiente?(itReg) do
                Encolar(infoJ.Rcambios, ⟨Siguiente(itReg), NomTab, False⟩)   O(L)
                Avanzar(itReg)                                               O(1) por ref
            end while
        end if
    end while
end if
Habiendo actualizado las colas temporales de los joins con los
registros que cumplen con el criterio de borrado
de los joins correspondientes.
Elimino del conjunto de registros, aquellos que cumplen el criterio de borrado.
Siendo n la cantidad de registros de t y T la cantidad de tablas en la base
En peor caso con costo O(n)
borrarRegistro(r, T_actual)                                                  O(T*L + n)
if CantidadDeAccesos(T)  b.TablaMaxima.#Modif then
    b.TablaMaxima.NomTabla ← Copiar(Nombre(T))                               O(L)
    b.TablaMaxima.#Modif ← Copiar(CantidadDeAccesos(T))
                                                                             O(1) por ref
end if                                                                       ---------------
                                                                             O(T + Log(n))
```

GENERARVISTAJOIN(**in** *t1* : `string`, *in t2* : `string`, *in c* : `campo`, *in/out b* : `estr`)

```
Cola(DatoCambio) Rcambio vacia()
Campo campoJ ←
Tipo campoT ← tipoCampo(c, Ta1)
DiccTrie(string; itConj(registro)) JoinS ← Vacio()
DiccNat(nat; itConj(registro)) JoinN ← Vacio()
Conj(Registro) JoinC ← vacio()                                               O(1) por ref
Tabla Ta1 ← Obtener(b.tablas, t1).Tactual                                    O(1) por ref
Tabla Ta2 ← Obtener(b.tablas, t2).Tactual                                    O(1) por ref
Nat n ←Cardinal(Registros(Ta1))                                              O(1) por ref
Nat m ←Cardinal(Registros(Ta2))                                              O(1) por ref
Conj(dato) cjd1←dameColumna(c, Registros(Ta1))                               O(nLog(n))
Conj(dato) cjd2←dameColumna(c, Registros(Ta2))                               O(mLog(m))
Ahora busco la interseccion de estos datos entre los conjuntos cjd1 y cjd2
ItConj(dato) itcjd1 ← CrearIt(cjd1)                                          O(1) por ref
DiccString(String, bool) ds ← vacio()                                        O(1) por ref
DiccNat(Nat, bool) dn ← vacio()                                              O(1) por ref
Conj(Dato) cjD ← vacio()
Cargo los valores de t1 en un dicc que depende del tipo del campo c
if ¬campoT then                                                              O(1) por ref
    El tipo del campo es string
    while HaySiguiente?(itcjd1) do                                           O(n)
        Sabemos que el dameColumna no tiene valores repetidos
        Definir(ds, ValorString(Siguiente(itcjd1)), true)                    O(1) por ref
        Avanzar(itcjd1)                                                      O(1) por ref
    end while
    ItConj(dato) itcjd2 ← CrearIt(cjd2)                                      O(1) por ref
    Ahora buscamos la interseccion
    while HaySiguiente?(itcjd2) do                                           O(m)
        Sabemos que el dameColumna no tiene valores repetidos
```

```
        Bool Def ← Definido?(ds, ValorString(Siguiente(itcjd2)))
                                                          O(1) por ref
        if Def then AgregarRapido(cjD, Siguiente(itcjd2))  O(1) por ref
        end if
        Avanzar(itcjd1)                                   O(1) por ref
    end while
else
    El tipo del campo es nat
    while HaySiguiente?(itcjd1) do                        O(n*log(n))
        Sabemos que el dameColumna no tiene valores repetidos
        Definir(dn, ValorNat(Siguiente(itcjd1)), true)    O(Log(n))
        Avanzar(itcjd1)                                   O(1) por ref
    end while
    ItConj(dato) itcjd2 ← CrearIt(cjd2)                   O(1) por ref
    Ahora buscamos la interseccion
    while HaySiguiente?(itcjd2) do                        O(m*Log(n))
        Sabemos que el dameColumna no tiene valores repetidos
        Bool Def ← Definido?(dn, ValorString(Siguiente(itcjd2)))
                                                          O(log(n))
        if Def then AgregarRapido(cjD,Siguiente(itcjd2))  O(1) por ref
        end if
        Avanzar(itcjd2)                                   O(1) por ref
    end while
end if
Si hay algun valor en la interseccion, con esos valores hago el join.
if ¬Vacio?(cjD) then                                      O(1) por ref
    Caso Strings
    its ← CrearIt(cjD)                                    O(1) por ref
    while HaySiguiente?(its) do                           O((n+m)*(L+Log(n*m)))
        Registro regModelo ← Definir(vacio(), c, Siguiente(its))
        Como el campo es clave en ambas tablas
        Si en ambas tablas hay indice en base a c y la complejidad es
        O(L+Log(n*m)) sino es O(cardinal de registros de la tabla)
        Asumimos que es el mejor caso para la complejidad.
        Registro r1 Siguiente(cj1) ← BuscarEnTabla(regModelo, ta1)
                                                          O(L+Log(n*m))
        Registro r2 Siguiente(cj1) ← BuscarEnTabla(regModelo, ta1)
                                                          O(L+Log(n*m))
        if ¬campoT then
            itConj(Registro) itnuevo AgregarRapido(JoinC, UnirRegistros(r1, r2))
                                                          O(1) por ref
            Definir(JoinS, ValorString(Siguiente(its)), itnuevo)  O(1) por ref
        else
            itConj(Registro) itnuevo AgregarRapido(JoinC, UnirRegistros(r1, r2))
                                                          O(1) por ref
            Definir(JoinN, ValorNat(Siguiente(its), itnuevo)      O(Log(n+m))
        end if
        Avanzar(its)                                      O(1) por ref
    end while
end if
info_join ← < Rcambios, campoJ, campoT, JoinS, JoinN, JoinC >
                                                          O(1) por ref
```

Definir(Ta1.Joins, t2, info_join) O(1) por ref
Definir(Ta2.Joins, t1, info_join) O(1) por ref

O(1) por ref

BORRARJOIN(**in** *t1* : `string`, *in t2* : `string`, *in/out b* : `estr`)
Tabla tab1 ← DameTabla(t1, b)
Tabla tab2 ← DameTabla(t2, b)
Borrar(tab1.Joins, t2)
Borrar(tab2.Joins, t1)

O(1) por ref

BUSCAR(**in** *criterio* : `registro`, *in t* : `string`, *in b* : `base`) ⟶ *res* : `conj(ItConj(registro))`
Busco los datos de la tabla.
info_tabla infot ← Significado(b.tablas, t) O(1) por ref
Tabla tab ← infot.TActual O(1) por ref
res ← BuscarEnTabla(criterio, tab)

O(1) por ref

REGISTROS(**in** *t* : `string`, *in b* : `base`) ⟶ *res* : `Conj(registros)`
$info \leftarrow Significado(b.tablas, t)$ O(1) por ref
$tab \leftarrow info.TActual$ O(1) por ref
$res \leftarrow registros(tab)$ O(1) por ref

O(1) por ref

VISTAJOIN(**in** *t1* : `string`, *in t2* : `string`, *in/out b* : `estr`)
info_tabla infot1 ← Significado(b.Tablas, t1) O(1) por ref
Tabla tab1 ← infot1.TActual O(1) por ref
info_tabla infot2 ← Significado(b.Tablas, t1) O(1) por ref
Tabla tab2 ← infot2.TActual O(1) por ref
info_join infoj ← Significado(infot1.Joins, t2) O(1) por ref
Campo c ← infoj.campoJ O(1) por ref
if ¬EsVacia?(infoj.Rcambios) **then** O(1) por ref
  Hubo cambios desde la generacion del join o del ultimo vistaJoin
  while ¬EsVacia?(infoj.Rcambios) **do**
    DatoCambio data ← Proximo(infoj.Rcambios) O(1) por ref
    Desencolar(infoj.Rcambios) O(1) por ref
    Registro r ← data.Reg O(1) por ref
    if data.Accion **then**
      La accion es agregar un registro al join
      Para hacer esto necesito saber a que tabla se agrego el registro
      NombreTabla NomTorigen ← data.NomOrigen O(1) por ref
      Sabiendo la tabla de origen, necesito identificar la otra tabla para ver si
      hay un registro con el mismo valor para el campo del join.
      Armo un registro auxiliar regModelo con el campo
      Dicc(campo,dato) regModelo ← vacio()
      Definir(regModelo, c, Significado(r, c))
      Si hay indice en la tabla para el campo c y ademas es campo clave
      BuscarEnTabla(tab, regModelo) tiene complejidad O(L) u O(Log(n))
      dependiendo del tipo del campo c.
      Si no hay indice para el campo c entonces BuscarEnTabla(tab, regModelo)
      tiene complejidad O(m) siendo m la cant de registros en tab.
      Registro rotro ← vacio()

```
            if NomTorigen=t1 then
                Entonces el otro registro lo tengo que buscar en t2
                Registro rotro ← Siguiente(BuscarEnTabla(tab2, regModelo))
            else
                Entonces el otro registro lo tengo que buscar en t1
                Registro rotro ← Siguiente(BuscarEnTabla(tab1, regModelo))
            end if
            if ¬Vacio?(rotro) then
                Si habia un registro que cumplia con lo pedido
                Registro rnuevo ← UnirRegistros(r, rotro)
                if info_join.campoT then
                    El tipo del campo es Natural.
                    itConj(registro) itnew ← AgregarRapido(info_join.JoinC, rnuevo)
                    Definir(info_join.JoinN, key, itnew)
                else
                    El tipo del campo es String.
                    itConj(registro) itnew ← AgregarRapido(info_join.JoinC, rnuevo)
                    Definir(info_join.JoinS, key, itnew)
                end if
            end if
        elseCasoBorrado
            if info_join.campoT then
                El tipo del campo es Natural.
                itConj(registro) itcjr ← Significado(info_join.JoinN, key)
                Borrar(info_join.JoinN, key)
                EliminarSiguiente(itcjr)
            else
                El tipo del campo es String.
                itConj(registro) itcjr ← Significado(info_join.JoinN, key)
                Borrar(info_join.JoinS, key)
                EliminarSiguiente(itcjr)
            end if
        end if
    end while
end if
res ← info_join.JoinC
```

O(Tamaño(infoj.Rcambios)*)

CANTIDADDEACCESOS(**in** t : `string`, *in* b : `base`) $\longrightarrow$ *res* : `nat`

$info \leftarrow Significado(b.tablas, t)$	O(1) por ref
$tab \leftarrow info.TActual$	O(1) por ref
$res \leftarrow cantidadDeAcccesos(tab)$	O(1) por ref
	O(1) por ref

TABLAMAXIMA(**in** b : `base`) $\longrightarrow$ *res* : `string`

$res \leftarrow b.TablaMaxima.NomTabla$	O(1) por ref
	O(1) por ref

8 Diccionario por Naturales

8.1 Interfaz

se explica con DICCIONARIO(NAT, σ)

usa Bool

géneros `diccNat(nat, σ)`

Operaciones

VACIO() $\longrightarrow$ *res* : `diccNat(nat, σ)`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} vacio()\}$
Descripción: Crea un nuevo diccionario
Complejidad: $O(1)$

DEFINIDO?(**in** d : `diccNat(nat, σ)` *in* n : `nat`) $\longrightarrow$ *res* : `Bool`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} def?(n, d)\}$
Descripción: Indica si la clave esta definida
Complejidad: $O(m)$

DEFINIR(**in/out** d : `diccNat(nat, σ)` *in* n : `nat`, *in* $s : \sigma$)
Pre $\equiv \{\neg def?(n, d) \land d = d_0\}$
Post $\equiv \{d =_{\text{obs}} Definir(n, s, d_0)\}$
Descripción: Se define s en el diccionario
Complejidad: $O(m)$

BORRAR(**in/out** d : `diccNat(nat, σ)` *in* n : `nat`)
Pre $\equiv \{d =_{\text{obs}} d_0 \land def?(n, d)\}$
Post $\equiv \{d =_{\text{obs}} Borrar(n, d_0)\}$
Descripción: Elimina el elemento n
Complejidad: $O(m)$

SIGNIFICADO(**in** d : `diccNat(nat, σ)` *in* n : `nat`) $\longrightarrow$ *res* : σ
Pre $\equiv \{def?(n, d)\}$
Post $\equiv \{res =_{\text{obs}} obtener(n, d)\}$
Descripción: Se retornan los significados
Complejidad: $O(m)$
Aliasing: Devuelve res por referencia.

DICCCLAVES(**in** d : `diccNat(nat σ)`) $\longrightarrow$ *res* : `itLista(nat)`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} claves(d)\}$
Descripción: Se retorna un iterador al primer elemento de la lista de claves del diccionario
Complejidad: $O(1)$

MAXIMO(**in** d : `diccNat(nat σ)`) $\longrightarrow$ *res* : `nat`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} max(claves(d))\}$
Descripción: Se retorna la clave maxima en forma de dato
Complejidad: $O(m)$

MINIMO(**in** d : `diccNat(nat` σ`)`) $\longrightarrow$ res : `nat`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} min(claves(d))\}$
Descripción: Se retorna la clave minima en forma de dato
Complejidad: $O(m)$

8.2 Representación

diccNat

se representa con `tupla`⟨dicc : `puntero(estr(nat` σ`))`,
claves : `lista(nat)`⟩

donde `estr(nat,` σ`)` **es** `tupla`⟨clave : `nat`,
significado : σ,
hijoDer : `puntero(estr(nat` σ`))`,
hijoIzq : `puntero(estr(nat` σ`))`,
itClaves : `itLista(nat)`⟩

donde `claves` **es** `Lista Enlazada del apunte de modulos basicos que contiene todas las claves del diccionario.`

donde `itClaves` **es** `Iterador bidireccional de lista claves que apunta al elemento correspondiente con su clave.`

Invariante de representación

Rep : $\widehat{\texttt{diccNat}} \longrightarrow boolean$

$(\forall e : \widehat{\texttt{diccNat}})$
$\text{Rep}(e) \equiv true \iff ((\forall n_1, n_2 : \text{estr(nat, } \sigma))(n_1 \in arbol(e.dicc) \wedge n_2 \in arbol(e.dicc) \Rightarrow_{\text{L}}$
$(n_1.clave < n_2.clave \Rightarrow n_1 \in arbol(n_2.hijoIzq)) \wedge (n_1.clave > n_2.clave \Rightarrow n_1 \in arbol(n_2.hijoDer)) \wedge$
$(\forall n_1, n_2 : \text{estr(nat, } \sigma))(n_1 \in arbol(e) \wedge n_2 \in arbol(e) \in n_1.clave \neq n_2.clave \Rightarrow_{\text{L}}$
$(n_1.hijoIzq = n_2.hijoIzq \vee n_1.hijoIzq = n_2.hijoDer \Rightarrow n_1.hijoIzq = NULL) \wedge$
$(n_1.hijoDer = n_2.hijoIzq \vee n_1.hijoDer = n_2.hijoDer \Rightarrow n_1.hijoDer = NULL) \wedge$
$((\forall n : \text{estr(nat, } \sigma))(n \in arbol(e.dicc)) \Rightarrow_{\text{L}} Esta?(e.claves, n.clave)) \wedge$
$((\forall k : \text{nat})(k \in e.claves) \Rightarrow (\exists n : estr(nat, \sigma))(n \in arbol(e) \wedge n.clave = k)))$

1. Para todo hijoDer de un estr, si no es NULL, su clave es mayor a la clave de su padre.
2. Para todo hijoIzq de un estr, si no es NULL, su clave es menor a la clave de su padre.
3. No hay ciclos, ni nodos con dos padres.
4. Todos las claves de los elementos del arbos estan en la lista claves y viceversa.

Función de abstracción

$\text{Abs} : \widehat{\texttt{diccNat(nat, } \sigma\texttt{)}}\ d \longrightarrow \widehat{\texttt{dicc(nat, } \sigma\texttt{)}}$ $\{\text{Rep}(d)\}$

$(\forall d : \widehat{\texttt{diccNat(nat, } \sigma\texttt{)}})$
$\text{Abs}(d) \equiv c : \widehat{\texttt{dicc(nat, } \sigma\texttt{)}} \mid ((\forall k : \text{nat})(k \in claves(c)) \Rightarrow$
$(\exists n : estr(nat, \sigma))(n \in arbol(d.dicc) \wedge n.clave = k) \wedge (k \in d.Claves)) \wedge$
$((\forall k : \text{nat})(k \in d.Claves) \Rightarrow k \in claves(c)) \wedge$
$((\forall n : \text{estr(nat, } \sigma))(n \in arbol(d.dicc) \Rightarrow n.clave \in claves(c)))) \wedge_{\text{L}}$
$((\forall n : \text{estr(nat, } \sigma))(n \in arbol(d.dicc) \Rightarrow obtener(c, n.clave) =_{\text{obs}} n.significado)$

$arbol : puntero(estr(nat, \sigma)) \leftarrow conj(puntero(estr(nat, \sigma)))$

$arbol(n) \equiv$
if $n.hijoIzq \neq null \wedge n.hijoDer \neq null$ **then**
$Ag(n, arbol(n.hijoIzq) \cup arbol(n.hijoDer))$
else
if $n.hijoIzq \neq null$ **then**
$Ag(n, arbol(n.hijoIzq))$
else
if $n.hijoDer \neq null$ **then**
$Ag(n, arbol(n.hijoDer))$
else
$Ag(n, \emptyset)$
end if
end if
end if

8.3 Algoritmos

iVacio() $\longrightarrow$ res : `estr`
$res \leftarrow NULL$

	O(1)

iDefinir(**in/out** d : `diccNat`, *in* n : `nat`, *in* $s : \sigma$)
$auxDefinir(\pi_1(d), \pi_2(d), n, s)$ O(m)

	O(m)

auxDefinir(**in/out** d : `estr`, *in* l : `lista(nat)`, **in** n : `nat`, *in* $s : \sigma$)
if $d == NULL$ **then**
$res \leftarrow < n, s, NULL, NULL, AgregarAtras(l, n) >$ O(1)
end if
if $d \neq NULL \wedge n < d.clave$ **then**
$d.hijoIzq \leftarrow iDefinir(d.hijoIzq, n, s)$ O(m)
end if
if $d \neq NULL \wedge n > d.clave$ **then**
$d.hijoDer \leftarrow iDefinir(d.hijoDer, n, s)$ O(m)
end if

	O(m)

iBorrar(**in/out** d : `diccNat`, *in* n : `nat`)
$auxBorrar(\pi_1(d), \pi_2(d), n, s)$ O(m)

	O(m)
auxBorrar(**in/out** d : `estr`, *in* n : `nat`)	
if $d == NULL$ **then**	
$FinFuncion$	O(1)
else if $n > d.clave$ **then**	
$d.hijoDer \leftarrow iBorrar(d.hijoDer, n)$	O(m)
else if $n < d.clave$ **then**	
$d.hijoIzq \leftarrow iBorrar(d.hijoIzq, n)$	O(m)
else if $d.hijoIzq == NULL \wedge d.hijoDer == NULL$ **then**	
$d.itClaves.Anterior.Siguiente \leftarrow d.itClaves.Siguiente$	O(1)
$d.itClaves.Siguiente.Anterior \leftarrow d.itClaves.Anterior$	O(1)
$Borrar(d)$	O(1)
$d \leftarrow NULL$	O(1)
else if $d.hijoIzq == NULL$ **then**	
$aux \leftarrow d.hijoDer$	O(1)
while $aux.hijoIzq \neq NULL$ **do**	O(m)
$aux \leftarrow aux.hijoIzq$	O(1)
end while	
$d.hijoDer \leftarrow iBorrar(aux.clave, d.hijoDer)$	
$d.clave \leftarrow aux.clave$	O(1)
$d.significado \leftarrow aux.significado$	O(1)
$d.itClaves \leftarrow aux.itClaves$	O(1)
else	
$aux \leftarrow d.hijoIzq$	O(1)
while $aux.hijoDer \neq NULL$ **do**	O(m)
$aux \leftarrow aux.hijoDer$	O(1)
end while	
$d.hijoIzq \leftarrow iBorrar(aux.clave, d.hijoIzq)$	
$d.clave \leftarrow aux.clave$	O(1)
$d.significado \leftarrow aux.significado$	O(1)
$d.itClaves \leftarrow aux.itClaves$	O(1)
end if	
	O(m)
iSignificado(**in/out** d : `diccNat`, *in* n : `nat`) $\longrightarrow$ $res : \sigma$	
$res \leftarrow auxSignificado(\pi_1(d), n)$	O(m)
	O(m)
auxSignificado(**in/out** d : `estr`, *in* n : `nat`) $\longrightarrow$ $res : \sigma$	
$nodoActual \leftarrow d$	O(1)
while $\neg(nodoActual == NULL) \wedge \neg res$ **do**	O(m)
if $nodoActual.clave == n$ **then**	
$res \leftarrow nodoActual.significado$	O(1)
else	
if $c < nodoActual.clave$ **then**	
$nodoActual \leftarrow nodoActual.hijoIzq$	O(1)
else	
$nodoActual \leftarrow nodoActual.hijoDer$	O(1)
end if	
end if	
end while	

	O(m)

iDefinido?(**in/out** d : `diccNat`, *in* n : `nat`) $\longrightarrow$ res : `bool`

$res \leftarrow auxDefinido?(\pi_1(d), n)$	O(m)
	O(m)

auxDefinido?(**in/out** d : `estr`, *in* n : `nat`) $\longrightarrow$ res : `bool`

$nodoActual \leftarrow d$	O(1)
$res \leftarrow FALSE$	O(1)
while $\neg(nodoActual == NULL) \wedge \neg res$ **do**	O(m)
if $nodoActual.clave == n$ **then**	
$res \leftarrow TRUE$	O(1)
else	
if $c < nodoActual.clave$ **then**	
$nodoActual \leftarrow nodoActual.hijoIzq$	O(1)
else	
$nodoActual \leftarrow nodoActual.hijoDer$	O(1)
end if	
end if	
end while	
	O(m)

iDiccClaves(**in/out** d : `diccNat`, *in* n : `nat`) $\longrightarrow$ res : `itLista(nat)`

$res \leftarrow CrearIt(\pi_2(d))$	
	O(m)

iMaximo(**in/out** d : `diccNat`) $\longrightarrow$ res : `nat`

if $\pi_1(d).hijoDer \neq NULL$ **then**	
$aux \leftarrow \pi_1(d).hijoDer$	O(1)
while $aux.hijoDer \neq NULL$ **do**	O(m)
$aux \leftarrow aux.hijoDer$	O(1)
end while	
$res \leftarrow aux.clave$	O(1)
else	
$res \leftarrow \pi_1(d).clave$	O(1)
end if	
	O(m)

iMinimo(**in/out** d : `diccNat`) $\longrightarrow$ res : `nat`

if $\pi_1(d).hijoIzq \neq NULL$ **then**	
$aux \leftarrow \pi_1(d).hijoIzq$	O(1)
while $aux.hijoIzq \neq NULL$ **do**	O(m)
$aux \leftarrow aux.hijoIzq$	O(1)
end while	
$res \leftarrow aux.clave$	O(1)
else	
$res \leftarrow \pi_1(d).clave$	O(1)
end if	
	O(m)

m: En peor caso es igual a la cantidad de elementos del arbol. En promedio es log(cantidad de elementos del arbol).

9 Modulo Diccionario Lexicografico(*String*, σ)

9.1 Interfaz

se explica con DICCIONARIO(STRING, σ) ITERADOR BIDIRECCIONAL (TUPLA(STRING, σ))

géneros `diccString(String, σ)`, `itDiccString(String, σ)`, `itClavesString`

Operaciones del diccionario

VACIO() $\longrightarrow$ *res* : `diccString(String, σ)`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} vacio()\}$
Descripción: Crea un nuevo diccionario
Complejidad: $O(1)$

DEFINIDO?(**in** *d* : `diccString(String, σ)` *in n* : `String`) $\longrightarrow$ *res* : `Bool`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} def?(n, d)\}$
Descripción: Indica si la clave esta definida
Complejidad: $O(Longitud(n))$

DEFINIR(**in/out** *d* : `diccString(String, σ)` *in n* : `String`, *in s* : σ)
Pre $\equiv \{d = d_0\}$
Post $\equiv \{d =_{\text{obs}} Definir(n, s, d_0)\}$
Descripción: Se define s en el diccionario
Complejidad: $O(Longitud(n))$
Aliasing: *s* se define por referencia. En caso de ya estar definido n, pisa la definición anterior

BORRAR(**in/out** *d* : `diccString(String, σ)` *in n* : `String`)
Pre $\equiv \{d =_{\text{obs}} d_0 \wedge def?(n, d)\}$
Post $\equiv \{d =_{\text{obs}} Borrar(n, d_0)\}$
Descripción: Elimina el elemento n
Complejidad: $O(Longitud(n))$

SIGNIFICADO(**in** *d* : `diccString(String, σ)` *in n* : `String`) $\longrightarrow$ *res* : σ
Pre $\equiv \{def?(n, d)\}$
Post $\equiv \{res =_{\text{obs}} obtener(n, d)\}$
Descripción: Se retorna el significado de n
Complejidad: $O(Longitud(n))$

DICCCLAVES(**in** *d* : `diccString(String σ)`) $\longrightarrow$ *res* : `conj(string)`
Pre $\equiv \{TRUE\}$
Post $\equiv \{res =_{\text{obs}} claves(d)\}$
Descripción: Se retorna el conjunto de claves del diccionario
Complejidad: $O(1)$
Aliasing: Res un conjunto devuelto por referencia no modificable.

MAXIMO(**in** *d* : `diccString(String σ)`) $\longrightarrow$ *res* : `String`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} max(claves(d))\}$
Descripción: Se retorna la clave maxima en orden lexicográfico
Complejidad: $O(longitud(k))$ donde k es la aplabra más larga del diccionario

MINIMO(**in** d : `diccString (String` σ`)`) $\longrightarrow$ res : `String`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}} min(claves(d))\}$
Descripción: Se retorna la clave minima en orden lexicográfico
Complejidad: $O(longitud(k))$ donde k es la aplabra más larga del diccionario

9.2 Operaciones del iterador

El iterador que presentamos permite modificar el diccionario recorrido. Sin embargo, cuando el diccionario es no modificable, no se pueden utilizar las funciones de eliminacion. Ademas, las claves de los elementos iterados no pueden modificarse nunca, por cuestiones de implementacion. Cuando d es modificable, decimos que it es modificable.

Para simplificar la notacion, vamos a utilizar clave y significado en lugar de Π_1 y Π_2 cuando utilicemos una tupla($String, \sigma$). CREARIT(**in** d : `diccString(String,` σ`)` *in* n : `String`) $\longrightarrow$ res : `itDiccString(`$String, \sigma$`)`
Pre $\equiv$ $\{true\}$
Post $\equiv$ {alias(esPermutacion(SecuSuby(res), d)) $\wedge$ vacia?(Anteriores(res))}
Descripción: crea un iterador bidireccional del diccionario, que apunta al primer elemento del mismo en orden lexicografico.
Complejidad: $O(CL * long(k))$ Donde CL es la cantidad de claves de d y k la palabra mas larga de d
Aliasing: hay aliasing entre los significados en el iterador y los del diccionario

HAYSIGUIENTE(**in** it : `itDiccString(String,` σ`)` *in* n : `String`) $\longrightarrow$ res : `bool`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}}$ haySiguiente?(it)$\}$
Descripción: devuelve `true` si y solo si en el iterador todavia quedan elementos para avanzar.
Complejidad: $O(1)$

HAYANTERIOR(**in** it : `itDiccString(String,` σ`)` *in* n : `String`) $\longrightarrow$ res : `bool`
Pre $\equiv$ $\{true\}$
Post $\equiv$ $\{res =_{\text{obs}}$ hayAnterior?(it)$\}$
Descripción: devuelve `true` si y solo si en el iterador todavia quedan elementos para retroceder.
Complejidad: $O(1)$

SIGUIENTE(**in** it : `itDiccString(String,` σ`)` *in* n : `String`) $\longrightarrow$ res : `tupla(String,` σ`)`
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Siguiente(it))}
Descripción: devuelve el elemento siguiente del iterador.
Complejidad: $O(1)$
Aliasing: res.significado es modificable si y solo si it es modificable, por aliasing. En cambio, res.clave no es modificable.

SIGUIENTECLAVE(**in** it : `itDiccString(String,` σ`)` *in* n : `String`) $\longrightarrow$ res : `String`
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Siguiente(it).clave)}
Descripción: devuelve la clave del elemento siguiente del iterador.
Complejidad: $O(1)$
Aliasing: res no es modficable.

SiguienteSignificado(**in** it : `itDiccString(String,` σ`)` in n : `String`) $\longrightarrow$ res : σ
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Siguiente(it).significado)}
Descripción: devuelve el significado del elemento siguiente del iterador.
Complejidad: $O(1)$
Aliasing: res es modificable si y solo si it es modficable, por aliasing.

Anterior(**in** it : `itDiccString(String,` σ`)` in n : `String`) $\longrightarrow$ *tupla(clave* : *String*, `significado:` σ`)`
Pre $\equiv$ {HayAnterior?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Anterior(it))}
Descripción: devuelve el elemento anterior del iterador.
Complejidad: $O(1)$
Aliasing: res.significado es modificable si y solo si it es modificable, por aliasing. En cambio, res.clave no es modificable.

AnteriorClave(**in** it : `itDiccString(String,` σ`)` in n : `String`) $\longrightarrow$ res : *String*
Pre $\equiv$ {HayAnterior?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Anterior(it).clave)}
Descripción: devuelve la clave del elemento anterior del iterador.
Complejidad: $O(1)$
Aliasing: res no es modficable.

AnteriorSignificado(**in** it : `itDiccString(String,` σ`)` in n : `String`) $\longrightarrow$ res : σ
Pre $\equiv$ {HayAnterior?(it)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Anterior(it).significado)}
Descripción: devuelve el significado del elemento anterior del iterador.
Complejidad: $O(1)$
Aliasing: res es modificable si y solo si it es modficable, por aliasing.

Avanzar(**inout** it : `itDiccString(String,` σ`)` in n : `String`)
Pre $\equiv$ {$it = it_0 \wedge$ HaySiguiente?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Avanzar(it_0)}
Descripción: avanza a la posicion siguiente del iterador.
Complejidad: $O(1)$

Retroceder(**inout** it : `itDiccString(String,` σ`)` in n : `String`)
Pre $\equiv$ {$it = it_0 \wedge$ HayAnterior?(it)}
Post $\equiv$ {$it =_{\text{obs}}$ Retroceder(it_0)}
Descripción: retrocede a la posicion anterior del iterador.
Complejidad: $O(1)$

9.3 Representacion

diccString

se representa con `dLex)`

donde `dLex` **es** `tupla`⟨raiz : `puntero(nodo)`,
claves : `conj(string)`⟩

nodo

se representa con `enodo)`

donde `enodo` **es** `tupla`⟨dato : σ,
esSig? : `Bool`,
claveEnConj : `itConj(String)`,
continuaciones : `puntero(char)` $\sqsubset 256 \sqsupset$⟩

1. conj(string) esel Conjunto Lineal de los modulos Básicos. itConj(string) es el iterador del mismo

Invariante de representación

1. Todo Nodo, si sus continuaciones son todos punteros a null, es porque es significado.
2. No hay ciclos, ni nodos con dos padres.
3. En el conjunto claves sólo se encuentran definidas las claves del diccionario y se encuentran todas ellas

Función de abstracción

Abs : diccString(string) d $\longrightarrow$ $dicc(String\sigma)$ {Rep (d)}

Abs(d) $\equiv$ AbsAux(d d.claves)

AbsAux : diccString(String) $dconj(String)/c$ $\longrightarrow$ dicc(String σ){$Rep(d) \wedge c \subseteq d.claves$}

AbsAux(d,c) $\equiv$ **if** $\emptyset?(c)$ **then**
$\emptyset$
else
$efinir(dameUno(c), significado(dameUno(c), d), AbsAux(d, sinUno(c)))$
fi

Respresentación del iterador

El iterador del diccionario lo recorre en orden lexicográfico. Los significados están por referencia.Se explica con el iterador bidireccional no modificable. itDiccString($String$, σ)

se representa con `itdLex)`

donde `dLex` **es** `tupla`⟨claves : `Lista(`$String$`)`,
significados : `Lista(`σ`)`⟩

9.4 Algoritmos

9.4.1 Algoritmos del Diccionario

iVacio() $\longrightarrow$ *res* : `diccString`	
$res.raiz \leftarrow NULL$	O(1)
$res.claves \leftarrow Vacio$	O(1)
	O(1)
iDefinir(**in/out** d : `diccString`, *in* n : `String`, *in* $s : \sigma$)	
$aux \leftarrow d.raiz$	O(1)
$i \leftarrow 0$	O(1)
while $i < Longitud(n)$ **do**	O(Longitud(n))
if $aux == NULL$ **then**	
$nNodo.esSig? \leftarrow false$	O(1)
$j \leftarrow 0$	
while $j < 256$ **do**	O(256)=O(1)
$nNodo.continuaciones[j] \leftarrow NULL$	O(1)
end while	
$aux \leftarrow \&nNodo$	O(1)
end if	
$aux \leftarrow aux.continuaciones[ord(n[i])]$	O(1)
end while	
$aux * .esSig? \leftarrow True$	O(1)
if $aux * .esSig? \neq True$ **then**	
$aux * .claveEnConj \leftarrow AgregarRapido(d.claves, n)$	O(long(n))
end if	
$aux * .esSig? \leftarrow True$	O(1)
$aux * .dato \leftarrow s$	O(1))
el último paso se realiza por referencia	
	O(Longitud(n))
iDefinido?(**in/out** d : `diccString`, *in* n : `String`) $\longrightarrow$ *res* : `bool`	
$aux \leftarrow d.raiz$	O(1)
$i \leftarrow 0$	O(1)
while $i < (Longitud(n) - 1) \wedge aux \neq NULL$ **do**	O(Longitud(n))
$aux \leftarrow aux.continuaciones[ord(n[i])]$	O(1)
end while	
if $aux \neq NULL$ **then**	
$res \leftarrow aux * .esSig?$	O(1)
else	
$res \leftarrow false$	O(1)
end if	
	O(Longitud(n))

iSignificado(**in** d : `diccString`, *in* n : `String`) $\longrightarrow$ *res* : σ

$aux \leftarrow d.raiz$	O(1)
$i \leftarrow 0$	O(1)
while $i < Longitud(n)$ **do**	O(Longitud(n))
$aux \leftarrow aux.continuaciones[ord(n[i])]$	O(1)
$i \leftarrow i + 1$	
end while	
$res \leftarrow aux * .dato$	O(1)(es una referencia)
	O(Longitud(n))

iBorrar(**in/out** d : `diccString`, *in* n : `String`)

$aux \leftarrow d.raiz$	O(1)
$i \leftarrow 0$	O(1)
$pila(puntero(nodo))p \leftarrow Vacia()$	O(1)
while $i < Longitud(n)$ **do**	O(Longitud(n))
Apilar(p,aux)	O(1)
$aux \leftarrow aux.continuaciones[ord(n[i])]$	O(1)
$i \leftarrow i + 1$	
end while	
$aux * .esSig? \leftarrow false$	
$BorrarSiguiente(aux * .claveEnConj)$	
$aux * esSig? \leftarrow false$	
$i \leftarrow i - 1$	
$j \leftarrow 0$	
while $aux * .continuaciones[j] = NULL \wedge j < 256$ **do**	O(256)=O(1)
$j \leftarrow j + 1$	
end while	
if $j < 256$ **then**	
$p \leftarrow Vacia()$	
end if	
while $\neg EsVacia?(p)$ **do**	
$j \leftarrow 0$	
$Tope(p) * .continuaciones[ord(n[i])] \leftarrow NULL$	
while $Tope(p) * .continuaciones[j] = NULL \wedge j < 256$ **do**	O(256)=O(1)
$j \leftarrow j + 1$	
end while	
if $j < 256$ **then**	
$p \leftarrow Vacia()$	
else	
$Desapilar(p)$	
$i \leftarrow i - 1$	
end if	
end while	
	O(Longitud(n))

iDiccClaves(**in/out** d : `diccString(String` σ`)`) $\longrightarrow$ *res* : $conj(String)$

$res \leftarrow d.claves$	
	O(1), d.claves se pasa por referencia

MAXIMO(**in/out** d : `diccString(String` σ`)`) $\longrightarrow$ res : $String$

```
aux ← d.raiz                                                    O(1)
res ← Vacia()
Boolf ← True
while f do
    nati ← 256
    while aux ∗ .continuaciones[i − 1] = NULL ∧ i > 0 do
        i ← i + 1
    end while
    if i = 0 then
        f ← false
    else
        AgregarAtras(res, ord⁻1(i − 1))
        i ← 0
    end if
end while
```

O(1) d.claves se pasa por referencia

MINIMO(**in/out** d : `diccString(String` σ`)`) $\longrightarrow$ res : $String$

```
aux ← d.raiz                                                    O(1)
res ← Vacia()
Boolf ← True
while f do
    nati ← 0
    while aux ∗ .continuaciones[i] = NULL ∧ i < 256 do
        i ← i + 1
    end while
    if i = 256 then
        f ← false
    else
        AgregarAtras(res, ord⁻1(i))
        i ← 0
    end if
end while
```

O(1) d.claves se pasa por referencia

9.4.2 Algoritmos del iterador

iCREARIT(**in/out** d : `diccString`, *in* n : `String`) $\longrightarrow$ res : `itDiccString`

```
lista(String)cs ← Vacia()
lista(σ)ss ← Vacia()
Stringn ← Vacio()
auxXrIt(cs, ss, n, d.raiz)                                      O(Longitud(k)*tam(d))
res.claves ← cs
res.significados ← ss
```

O(Longitud(k)*tam(d))

1. k es la palabra más larga definida en d y tam(d) es la cantidad de palabras definidas que hay [1].

[1] Si bien no nos fue posible realizar el calculo de complejidad correspondiente, el algoritmo recursivo recorre una vez cada nodo, y el total de nodos esta acotado por la sumatoria del largo de cada palabra, que a su vez está acotada por la cantidad de palabras multiplicada por la longitud de la palabra más larga, por lo que la cota propuesta a la complejidad es razonable

auxCrIt(**in/out** cs : `Lista(string)` **in/out** `ss:Lista(`σ`)` **in/out** `n:String` **in** `p:puntero(nodo)`)

if $p \neq NULL$ **then**
  if $p*.esSig?$ **then**
    $AgregarAtras(cs, n)$
    $AgregarAtras(ss, p*.dato)$ — O(1)
  end if
  $i \leftarrow 0$
  while $i < 256$ **do**
    $AgregarAtras(n, ord^{-}1(i))$
    $auxCrIt(cs, ss, n, p*.continuaciones[i])$
    $TirarUltimos(n, 1)$
  end while
end if

Tn desconocido

iHaySiguiente(**in/out** it : `itDiccString`, $in\ n$: `String`) $\longrightarrow$ res : $bool$

$res \leftarrow it.claves.siguiente \neq NULL$

O(1)

iHayAnterior(**in/out** it : `itDiccString`, $in\ n$: `String`) $\longrightarrow$ res : $bool$

$res \leftarrow it.claves.anterior \neq NULL$

O(1)

iSiguiente(**in/out** it : `itDiccString`, $in\ n$: `String`) $\longrightarrow$ res : `tupla(string,`σ`)`

$res.clave \leftarrow it.claves.siguiente*.dato$ — O(1)(es una referencia)

$res.significado \leftarrow it.dignificados.siguiente*.dato$ — O(1)(es una referencia)

O(1)

iSiguienteclave(**in/out** it : `itDiccString`, $in\ n$: `String`) $\longrightarrow$ res : $String$

$res \leftarrow it.claves.siguiente*.dato$ — O(1)(es una referencia)

O(1)

iSiguienteSignificado(**in/out** it : `itDiccString`, $in\ n$: `String`) $\longrightarrow$ res : σ

$res \leftarrow it.significados.siguiente*.dato$ — O(1)(es una referencia)

O(1)

iAnterior(**in/out** it : `itDiccString`, $in\ n$: `String`) $\longrightarrow$ res : `tupla(string,`σ`)`

$res.clave \leftarrow it.claves.anterior*.dato$ — O(1)(es una referencia)

$res.significado \leftarrow it.significados.anterior*.dato$ — O(1)(es una referencia)

O(1)

iAnteriorClave(**in/out** it : `itDiccString`, $in\ n$: `String`) $\longrightarrow$ res : $String$

$res \leftarrow it.claves.anterior*.dato$ — O(1)(es una referencia)

O(1)

iAnteriorSignificado(**in/out** it : `itDiccString`, $in\ n$: `String`) $\longrightarrow$ res : σ

$res \leftarrow it.significados.anterior*.dato$ — O(1)(es una referencia)

O(1)

iAvanzar(**in/out** *it* : `itDiccString`, *in* *n* : `String`)	
$it.claves \leftarrow it.claves.siguiente$	O(1)(es una referencia)
$it.significados \leftarrow it.significados.siguiente$	O(1)(es una referencia)
	O(1)
iRetroceder(**in/out** *it* : `itDiccString`, *in* *n* : `String`)	
$it.claves \leftarrow it.claves.anterior$	O(1)(es una referencia)
$it.significados \leftarrow it.significados.anterior$	O(1)(es una referencia)
	O(1)