DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2: Diseño

Primer cuatrimestre - 2016 Algoritmos y Estructuras de Datos II

Grupo XXXX

Integrante	LU	Correo electrónico
BENZO, Mariano	198/14	marianobenzo@gmail.com
FARIAS, Mauro	821/13	farias.mauro@hotmail.com
GUTTMAN, Martin	686/14	haris@live.com.ar
MOSQUEIRA C., Edgardo Ramon	808/13	edgarcab666@hotmail.com

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 Tabla

1.1 Interfaz

se explica con TABLA

usa

géneros `nat`, `dato`, `campo`, `tipo`, `registro`, `conjTrie`, `string`, `diccTrie(string, alfha)`, `diccAVL(`

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 : `itConjTrie(campo)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} claves(t)\}$
Descripción: Devuelve un conjunto de campos que son claves en la tabla ingresada por parametro.
Complejidad: O(1)
Aliasing: Se devuelve un iterador al conjunto claves por referencia.

INDICES(**in** t : `tab`) $\longrightarrow$ res : `itConjTrie(campo)`
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($calcular$)
Aliasing: Se devuelve res por referencia y no es modificable.

CAMPOS(**in** t : `tab`) $\longrightarrow$ 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.

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 ingresada por parametro.
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 ingresada por parametro.
Complejidad: O($L + log(n)$)
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* : `conjTrie(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($calcular$)

AGREGARREGISTRO(**in** r : `registro`, *in* 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($L + in$)
Aliasing: Agrega el registro r por referencia.

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

INDEXAR(**in** *crit* : `registro`, *in* t : `tab`)
Pre $\equiv$ {t_0=t $\wedge$ puedeIndexar(c,t)}
Post $\equiv$ {indexar(c,t_0)}
Descripción: Borra los registros que cumplan el criterio pasado por parametro.
Complejidad: O($L + in$)

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($T * L + in$)

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)

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($L + in$)
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: $O(L + in)$
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: $O(L + in)$

COINCIDENCIAS(**in** r : `registro`, *in* cj : `Conj(registro)`) $\longrightarrow$ *res* : `Conj(registro)`
Pre $\equiv \{True\}$
Post $\equiv \{res =_{\text{obs}} coincidencias(r, cj)\}$
Descripción: Compara el valor del registro con el conjunto de registros y retorna la interseccion.
Complejidad: $O(L + in)$
Aliasing: Retorna res por referencia.

HAYCOINCIDENCIA(**in** r : `registro`, **in** *cj1* : `ConjTrie(campo)`, *in cj2* : `Conj(registro)`) $\longrightarrow$ *res* : `bool`
Pre $\equiv \{True\}$
Post $\equiv \{res =_{\text{obs}} hayCoincidencia(r, cj1, cj2)\}$
Descripción: Compara los valores del registro para los campos dados por parametro, con el conjunto de registros.
Complejidad: $O(L + in)$

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: $O(L + in)$
Aliasing: Retorna res por copia.

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: $O(T * L + in)$
Aliasing: Retorna res por referencia.

MISMOSTIPOS(**in** r : `registro`, **in** t : `tab`) $\longrightarrow$ *res* : `bool`
Pre $\equiv \{\text{campos(r)}\subseteq\text{campos(t)}\}$
Post $\equiv \{res =_{\text{obs}} mismosTipos(r, t)\}$
Descripción: Compara los tipos correspondientes a los campos del registro y la tabla.
Complejidad: $O(1)$

1.2 Representación

se representa con `Tabla`
donde `tab` **es** `tupla`⟨Nombre : `String`,
Indices : `DiccTrie(campo, Indice)`,
Registros : `Conj(Registro)`,
Campos : `DiccTrie(Campo,Tipo)`,
#Accesos : `Nat`⟩
`donde Indice es tupla`⟨PorNat : `DiccTrie(Dato Conj(Registro))`,
PorString : `DiccTrie(Dato Conj(Registro))`⟩

Invariante de representación

1. Para todos los registros de r, el tipo de los datos de las columnas de r, deben coincidir con los tipos de las columnas en e.campos.
2. Todas las columnas de e.campos y su tipo, deben coincidir con los campos y tipo de todos los registros de e.registros. Es decir no debe haber campos de mas.'
3. El nombre de la tabla que figura en e.nombre, es un string de longitut acotada.
4. Para todo registro r de e.registros y para todo campo c de e.Indices.DiccClaves, se debe cumplir que si tenemos valor ← Obtener(r, c) y ind ← Obtener(e.Indices, c). Al evaluar que r ∈ Obtener(ind, valor) y deben ser del mismo tipo.
5. Para todo campo c, que pertenece a e.Indices.DiccClaves, si tenemos que ind ← Obtener(e.Indices, c), y para todo dato d perteneciente a ind.DiccClaves entonces Obtener(ind, d) esta incluido o es igual a e.registros.
6. Para todo registro r perteneciente a e.registros r.DiccClaves es igual a e.campos.DiccClaves.
7. El valor de e.#Accesos debe ser la cantidad de registros agregados, la cantidad de registros borrados, mas la cantidad de indices creados.

Función de abstracción

$\text{Abs} : \widehat{\texttt{sistema}}\ s \longrightarrow \widehat{\texttt{CampusSeguro}}$ $\{\text{Rep}(s)\}$

$(\forall s : \widehat{\texttt{sistema}})$
$\text{Abs}(s) \equiv cs : \widehat{\texttt{CampusSeguro}} \mid s.campus =_{\text{obs}} campus(cs) \wedge$
$s.estudiantes =_{\text{obs}} estudiantes(cs) \wedge$
$s.hippies =_{\text{obs}} hippies(cs) \wedge$
$s.agentes =_{\text{obs}} agentes(cs) \wedge$
$((\forall n : \text{nombre}) s.hippies.definido(n) \Rightarrow_{\text{L}} s.hippies.obtener(n) =_{\text{obs}} posEstYHippie(n, cs) \vee$
$(\forall n : \text{nombre}) s.estudiantes.definido(n) \Rightarrow_{\text{L}} s.estudiantes.obtener(n) =_{\text{obs}} posEstYHippie(n, cs))$
$(\forall pl : \text{placa}) s.agentes.definido(pl) \Rightarrow_{\text{L}} s.estudiantes.obtener(pl).pos =_{\text{obs}} posAgente(pl, cs))$
$(\forall pl : \text{placa}) s.agentes.definido(pl) \Rightarrow_{\text{L}} s.estudiantes.obtener(pl).cantSanciones =_{\text{obs}} cantSanciones(pl, cs))$
$(\forall pl : \text{placa}) s.agentes.definido(pl) \Rightarrow_{\text{L}} s.estudiantes.obtener(pl).cantCapturas =_{\text{obs}} cantCapturas(pl, cs))$

1.3 Algoritmos

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

$res \leftarrow t.nombre$	O(1)
	O(1)

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

$res \leftarrow t.Campos.ClavesDicc$	O(1)
	O(1)

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

$res \leftarrow t.ClavesDicc$	O(1)
	O(1)

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

$res \leftarrow t.Campos.ClavesDicc$	O(1)
	O(1)

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

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

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

$res \leftarrow t.registros$	θ(L+ log(n))
	θ(L+ log(n))

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

$res \leftarrow t.cantDeAccesos$	$\theta(1)$
	$\theta(1)$

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

itcampos $\leftarrow$ crearItTrie(Campos(columnas))	O(1)
res $\leftarrow$ $< nombre\, Vacio()\, Vacio()\, Vacio()\, 0 >$	O(1)
while HaySiguiente(itcampos) **do**	O(1)
Esto se debe a que # de campos a iterar es acotada.	
valor $\leftarrow$ Significado(r, Siguiente(itcampos))	O(1)
DefinirRapido(res.Campos, Siguiente(itcampos), valor)	O(1)
if Pertenece?(claves,Siguiente(itcampos)) **then**	O(1)
val $\leftarrow$ Significado(r, Siguiente(itcampos))	
AgregarRapido(res.Campos.CamposClave, val)	O(1)
end if	
Avanzar(itcampos)	O(1)
end while	
	$\theta(1)$

AGREGARREGISTRO(**in** r : `registro`, *in* t : `tab`)

nuevo $\leftarrow$ AgregarRapido(t.Registros,r)	$\theta(1)$
t.#Accesos++	$\theta(1)$
if Cardinal(t.Indices.ClavesDicc)$\geq$1 **then**	$\theta(1)$

itInd ← crearItConjTrie(t.Indices.ClavesDicc)	$\theta(1)$
while HaySiguiente(itInd) **do**	$\theta(1)$
indiceC ← Obtener(t.Indices, Siguiente(itInd))	$\theta(1)$
valorC ← Obtener(r, Siguiente(itInd))	$\theta(1)$
AgregarRapido(Obtener(indiceC, valorC), nuevo)	$\theta(1)$
Avanzar(itInd)	$\theta(1)$
end while	
end if	
	$\theta(1)$

BORRARREGISTRO(**in** *crit* : `registro`, *in* *t* : `tab`)

c ← Siguiente(Campos(crit))	$\theta(1)$
valor ← Obtener(crit, c)	$\theta(1)$
if Definido?(t.Indices, c) **then**	$\theta(1)$
indiceC ← Obtener(t.Indices, c)	$\theta(1)$
itcjr ← CrearItConj(Obtener(indiceC, valor))	$\theta(1)$
while HaySiguiente(itcjr) **do**	$\theta(\text{Log(n)})$
EliminarSiguiente(Siguiente(itcjr))	$\theta(1)$
tiene sentido???	
EliminarSiguiente(itcjr)	$\theta(1)$
end while	
else	
cr ← Coincidencias(crit, t.registros)	$\theta(\text{Cardinal(t.registros)})$
while HaySiguiente(cr) **do**	
EliminarSiguiente(Siguiente(cr))	
tiene sentido???	
EliminarSiguiente(cr)	
end while	
end if	
	$\theta(\text{Calcular despues de consulta})$

INDEXAR(**in** *c* : `campo`, *in* *t* : `tab`)

if tipoCampo(c,t) **then**	
conjLog(registro) nuevo ← vacio()	
else	
conjTrie(registro) nuevo ← vacio()	
end if	
indC ← Siguiente(DefinirRapido(t.Indices, c, nuevo))	
cr ← t.registros	
while HaySiguiente(cr) **do**	
valor ← Obtener(Siguiente(cr), c)	
if Definido?(indC, valor) **then**	
regviejos ← Obtener(indC, valor)	
AgregarRapido(regviejos, Siguiente(cr))	
else	
DefinirRapido(indC, valor, Siguiente(cr))	
end if	
Avanzar(cr)	
end while	
	$\theta(1)$

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

res ← campatible(r,t) ∧ ¬hayCoincidencia(r, r.ClavesDicc, registros(t)) — θ(L+log(n))

θ(L+log(n))

COMPATIBLE(**in** *r* : `registro`, *int t* : `tab`) ⟶ *res* : `bool`

res ← campatible(r,t) $\wedge_{\text{L}}$ mismosTipos(r,t) — θ(1)

O(1)

PUEDEINDEXAR(**in** *c* : `campo`, *in t* : `tab`) ⟶ *res* : `bool`

res ← Definido?(t.campos,c) $\wedge_{\text{L}} \neg$ Definido?(t.Indices,c) ∧ (Cardinal(t.Indices)≤1

O(calcular)

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

itcr1 ← CrearItConjTrie(cr1)	θ(1)
copiacr2 ← Copiar(cr2)	θ(Cardinal(cr2))
while HaySiguiente(itcr1) **do**	θ(Cardinal(cr1))
combinarTodos(c,Siguiente(itcr1),copiacr2)	θ(1)
Avanzar(itcr1);	θ(1)
end while	
res ← copiacr2;	θ(1)

O(Cardinal(cr1))

HAYCOINCIDENCIA(**in** *r* : `registro`, *in cc* : `ConjTrie(campo)`, *in cr* : `Conj(registro)`) ⟶ *res* : `bool`

itcr ← CrearItConj(cr);	θ(1)
res ← false;	θ(1)
while HaySiguiente(itcr) **do**	θ(Cardinal(cr))
res ← coincideAlguno(r,cc,Siguiente(itcr)) ∨res;	θ(1)
Avanzar(itcr);	θ(1)
end while	

O(Cardinal(cr))

COINCIDENCIAS(**in** *crit* : `registro`, *in cr* : `Conj(registro)`) ⟶ *res* : `Conj(registro)`

res ← Vacio();	θ(1)
itcr ← CrearItConj(cr)	
while HaySiguiente(cr) **do**	θ(Cardinal(cr))
if coincidenTodos(crit,campos(crit),Siguiente(itcr)) **then**	θ(1)
AgregarRapido(res,Siguiente(itcr))	θ(1)
end if	
Avanzar(itcr);	θ(1)
end while	

O(Cardinal(cr))

MINIMO(**in** *c* : `campo`, *in t* : `tab`) ⟶ *res* : `dato`

res ← min(dameColumna(c, t.registros)) — θ(Cardinal(t.registros))

O(Cardinal(t.registros))

MAXIMO(**in** *c* : `campo`, *in t* : `tab`) ⟶ *res* : `dato`

res ← max(dameColumna(c, t.registros)) — θ(Cardinal(t.registros))

	O(Cardinal(t.registros))

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

itcr ← CrearItConj(cr);	$\theta(1)$
res ← vacio();	$\theta(1)$
while HaySiguiente(itcr) **do**	θ(Cardinal(cr))
if ¬Pertenece(res, Siguiente(itcr)) **then**	θ(?????)
AgregarRapido(res, Siguiente(itcr))	
end if	
Avanzar(itcr);	$\theta(1)$
end while	
	O(calcular)

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

res ← True;	$\theta(1)$
itconjClaves ← CrearItConj(r.ClavesDicc);	$\theta(1)$
while HaySiguiente(itconjClaves) **do**	$\theta(1)$
val1← tipo?(Obtener(r,Siguiente(itconjClaves)))	θ(Cardinal(t.registros))
val2← tipoCampo(Siguiente(itconjClaves)),t)	$\theta(1)$
res ← res ∧ val1 = val2	$\theta(1)$
Avanzar(cr);	$\theta(1)$
end while	
	O(calcular)

1.4 Algoritmos operaciones auxiliares

2 Diccionario por Naturales

2.1 Interfaz

se explica con DICCIONARIO(κ, σ)

usa Bool

géneros `diccNat(`κ, σ`)`

Operaciones

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

DEFINIDO?(**in** d : `diccNat(`κ, σ`)` *in* $n : \kappa$) $\longrightarrow$ *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(`κ, σ`)` *in* $n : \kappa$, *in* $s : \sigma$)
Pre $\equiv \{\neg def?(n, d) \wedge d = d_0\}$
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(`κ, σ`)` *in* $n : \kappa$)
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: $O(m)$

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

CLAVES(**in** d : `diccNat(`$\kappa\sigma$`)`) $\longrightarrow$ res : `conjLog(`κ`)`
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} claves(d)\}$
Descripción: Se retornan el conjunto de claves del diccionario
Complejidad: $O(1)$

2.2 Representación

diccNat

se representa con `raiz: puntero(estr(`κ σ`))`

donde `estr(`κ`,` σ`)` **es** `tupla`⟨clave : κ,
significado : σ,
hijoDer : `puntero(estr(`$\kappa\sigma$`))`,
hijoIzq : `puntero(estr(`$\kappa\sigma$`))`,
diccClaves : `conjLog(`κ`)`⟩

Invariante de representación

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

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

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 diccClaves y viceversa.

Función de abstracción

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

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

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

$arbol(n) \equiv$
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

2.3 Algoritmos

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

	O(1)

iDefinir(**in/out** d : `estr`, *in* $n : \kappa$, *in* $s : \sigma$)
  if $d == NULL$ **then**
    $res \leftarrow < n, s, NULL, NULL >$ — O(1)
  end if
  if $d \neq NULL \wedge n < d.clave$ **then**
    $d.hijoIzq \leftarrow iDefinir(d.hijoIzq, n, s)$ — O(log(m))
  end if
  if $d \neq NULL \wedge n > d.clave$ **then**
    $d.hijoDer \leftarrow iDefinir(d.hijoDer, n, s)$ — O(m)
  end if
  $AgregarRapido(d.diccClaves, n)$ — O(m)

	O(m)

iBorrar(**in/out** d : `estr`, *in* $n : \kappa$)
  if $d == NULL$ **then**
    $FinFuncion$ — O(1)

```
else if n > d.clave then
    d.hijoDer ← iBorrar(d.hijoDer, n)                                   O(m)
else if n < d.clave then
    d.hijoIzq ← iBorrar(d.hijoIzq, n)                                   O(m)
else if d.hijoIzq == NULL ∧ d.hijoDer == NULL then
    Borrar(d)                                                           O(1)
    d ← NULL                                                            O(1)
else if d.hijoIzq == NULL then
    aux ← d.hijoDer                                                     O(1)
    while aux.hijoIzq ≠ NULL do                                         O(m)
        aux ← aux.hijoIzq                                               O(1)
    end while
    d.hijoDer ← iBorrar(aux.clave, d.hijoDer)
    d.clave ← aux.clave                                                 O(1)
    d.significado ← aux.significado                                     O(1)
else
    aux ← d.hijoIzq                                                     O(1)
    while aux.hijoDer ≠ NULL do                                         O(m)
        aux ← aux.hijoDer                                               O(1)
    end while
    d.hijoIzq ← iBorrar(aux.clave, d.hijoIzq)
    d.clave ← aux.clave                                                 O(1)
    d.significado ← aux.significado                                     O(1)
end if
Eliminar(d.diccClaves, n)                                               O(m)
                                                                        ---
                                                                        O(m)
```

iSignificado(**in/out** d : estr, *in* $n : \kappa$) $\longrightarrow$ $res : \sigma$

```
nodoActual ← d                                                          O(1)
while ¬(nodoActual == NULL) ∧ ¬res do                                   O(m)
    if nodoActual.clave == n then
        res ← nodoActual.significado                                    O(1)
    else
        if c < nodoActual.clave then
            nodoActual ← nodoActual.hijoIzq                             O(1)
        else
            nodoActual ← nodoActual.hijoDer                             O(1)
        end if
    end if
end while
                                                                        ---
                                                                        O(m)
```

iDefinido?(**in/out** d : estr, *in* $n : \kappa$) $\longrightarrow$ res : bool

```
nodoActual ← d                                                          O(1)
res ← FALSE                                                             O(1)
while ¬(nodoActual == NULL) ∧ ¬res do                                   O(m)
    if nodoActual.clave == n then
        res ← TRUE                                                      O(1)
    else
        if c < nodoActual.clave then
            nodoActual ← nodoActual.hijoIzq                             O(1)
        else
```

```
            nodoActual ← nodoActual.hijoDer                    O(1)
        end if
    end if
end while
```

O(m)

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

3 Modulo Diccionario Lexicografico(*String*, σ)

3.1 Interfaz

se explica con DICCIONARIO(STRING, σ)

usa Bool, Nat

géneros `diccString(String, σ)`

Trabajo Práctico 2: DiseñoOperaciones basicas de 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 $\equiv \{\neg def?(n, d) \land d = d_0\}$
Post $\equiv \{d =_{\text{obs}} Definir(n, s, d_0)\}$
Descripción: Se define s en el diccionario
Complejidad: $O(Longitud(n) + copy(s))$
Aliasing: *s* se define por copia

BORRAR(**in/out** *d* : `diccString(String, σ)` *in n* : `String`)
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: $O(Longitud(n))$

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: $O(Longitud(n))$

Trabajo Práctico 2: DiseñoOperaciones del iterador

El iterador que presentamos permite modificar el diccionario recorrido, eliminando elementos. 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, σ)`
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. 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.

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. 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.

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: $O(1)$

3.2 Representacion

diccString

se representa con `dLex)`

donde `dLex` **es** `tupla`⟨raiz : `puntero(nodo)`⟩

nodo

se representa con `enodo)`

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

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.

Función de abstracción

Respresentación del iterador

Trabajo Práctico 2: DiseñoRepresentacion del iterador

El iterador del diccionario lo recorre en orden lexicográfico. Los significados están por referencia.
itDiccString(*String*, σ)

se representa con `itdLex)`

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

3.3 Algoritmos

iVacio() $\longrightarrow$ *res* : `diccString`

$res.raiz \leftarrow NULL$	O(1)
$res.cardinal \leftarrow NULL$	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)
$aux*.dato \leftarrow s$	O(copy(s)) el costo de la copia de σ
	O(Longitud(n)+copy(s))

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 ← aux.continuaciones[ord(n[i])]                    O(1)
    i ← i + 1
  end while
  res ← aux ∗ .dato                                        O(1)(es una referencia)
                                                           ---
                                                           O(Longitud(n))
```

```
iBorrar(in/out d : diccString, in n : String)
  aux ← d.raiz                                             O(1)
  i ← 0                                                    O(1)
  pila(puntero(nodo))p ← Vacia()                           O(1)
  while i < Longitud(n) do                                 O(Longitud(n))
    Apilar(p,aux)                                          O(1)
    aux ← aux.continuaciones[ord(n[i])]                    O(1)
    i ← i + 1
  end while
  aux ∗ esSig? ← false
  i ← i − 1
  j ← 0
  while aux ∗ .continuaciones[j] = NULL ∧ j < 256 do       O(256)=O(1)
    j ← j + 1
  end while
  if j < 256 then
    p ← Vacia()
  end if
  while ¬EsVacia?(p) do
    j ← 0
    Tope(p) ∗ .continuaciones[ord(n[i])] ← NULL
    while Tope(p)∗.continuaciones[j] = NULL∧j < 256 do     O(256)=O(1)
      j ← j + 1
    end while
    if j < 256 then
      p ← Vacia()
    else
      Desapilar(p)
      i ← i − 1
    end if
  end while
                                                           ---
                                                           O(Longitud(n))
```

```
iCrearIt(in/out d : diccString, in n : String) ⟶ 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)) Donde k es la palal
```

4 Base de Datos

4.1 Interfaz

se explica con BASE

usa

géneros `nat`, `string`, `tabla`, `regisro`, `campo`, `dato`

Operaciones

TABLAS(**in** b : `base`) $\longrightarrow$ *res* : `conj(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.

DAMETABLA(**in** b : `base`) $\longrightarrow$ *res* : `tabla`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} claves(t)\}$
Descripción: Devuelve un conjunto de campos que son claves en la tabla ingresada por parametro.
Complejidad: O(1)
Aliasing: Se devuelve un iterador al conjunto claves 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($calcular$)
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($calcular$)

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($calcular$)
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 ≡ {insertarEntrada(r t b_0)}
Descripción: Inserta el registro a la tabla que corresponde al string pasado por parametro.
Complejidad: O(*calcular*)

BORRAR(**in** *cr* : `registro`, *in t* : `string`, *in b* : `base`)
Pre ≡ {b_0=b ∧ t∈tablas(b) ∧ #(cr.DiccClaves)}
Post ≡ {borrar(cr t b_0)}
Descripción: Borra los registros que cumplan el criterio cr pasado por parametro.
Complejidad: O(*calcular*)

GENERARVISTAJOIN(**in** *t1* : `string`, *in t2* : `string`, *in c* : `campo`, *in b* : `base`)
Pre ≡ {b_0=b ∧ t1⊐t2 ∧ {t1 t2}⊆tablas(b) ∧$_{\text{L}}$ (c∈dameTabla(t1 b).diccClaves ∧ c∈dameTabla(t2 b).diccCla
Post ≡ {generarVistaJoin(cr, t, b_0)}
Descripción: Borra los registros que cumplan el criterio cr pasado por parametro.
Complejidad: O(*calcular*)

BORRARJOIN(**in** *t1* : `string`, *in t2* : `string`, *in b* : `base`)
Pre ≡ {b_0=b ∧ hayJoin?(t1 t2 b)}
Post ≡ {borrarJoin(t1 t2 b_0)}
Descripción: Borra correspondiente a los nombres de tablas, pasados por parametro.
Complejidad: O(*calcular*)

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

VISTAJOIN(**in** *t1* : `string`, *in t2* : `string`, *in b* : `base`) ⟶ *res* : `conj(registro)`
Pre ≡ {{t1 t2}⊆tablas(b) ∧ hayJoin?(t1 t2 b)}
Post ≡ {$res =_{\text{obs}} vistaJoin(t1\,t2\,b)$}
Descripción: Retorna el conjunto de registros correspondientes al nombre de tabla pasado por parametro
Complejidad: O(*calcular*)
Aliasing: Se retorna el conjunto de registros por referencia.

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

TABLAMAXIMA(**in** *b* : `base`) ⟶ *res* : `string`
Pre ≡ {¬∅?(tablas(b))}
Post ≡ {$res =_{\text{obs}} tablaMaxima(t\,b)$}
Descripción: Retorna el nombre de la tabla con la mayor cantidad de modificaciones.
Complejidad: O(*calcular*)
Aliasing: Se retorna el nombre de la tabla por referencia.

ENCONTRARMAXIMO(**in** *t* : `string`, *in ct* : `conj(string)`, *in b* : `base`) ⟶ *res* : `string`
Pre ≡ {{t}∪ct ⊆tablas(b)}
Post ≡ {$res =_{\text{obs}} tablaMaxima(t\,b)$}

Descripción: Retorna ...
Complejidad: O($calcular$)
Aliasing: Se retorna el nombre de la tabla por referencia.

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

4.2 Representación

se representa con `Base`
donde `estr` **es** `tupla`$\langle$Tablas : `DiccTrie(Campo,info_tabla)`$\rangle$
`donde info_tabla es tupla`$\langle$#Accesos : `nat`,
TActual : `tabla`,
Joins : `DiccTrie(string info_join)`$\rangle$
`donde info_join es tupla`$\langle$R : `nat`,
Rborrados : `Conj(registro)`,
Ragregados : `Conj(registro)`,
campoJ : `campo`,
campoT : `tipo`,
Join : `Conj(registro)`$\rangle$

Invariante de representación

1. El Nombre de la tabla es un String acotado.
2. Indices es un arreglo de tamaño 2, que aloja el Indice correspondiente segun el orden de creacion.
3. Para toda Dato que es clave en Indice, su significado llamemoslo sign esta incluido en Registros.
4.

Función de abstracción

Abs : $\widehat{\texttt{sistema}}$ $s \longrightarrow \widehat{\texttt{CampusSeguro}}$ {Rep(s)}

($\forall s : \widehat{\texttt{sistema}}$)
Abs(s) $\equiv$ cs : $\widehat{\texttt{CampusSeguro}}$ | $s.campus =_{\text{obs}} campus(cs) \wedge$
$s.estudiantes =_{\text{obs}} estudiantes(cs) \wedge$
$s.hippies =_{\text{obs}} hippies(cs) \wedge$
$s.agentes =_{\text{obs}} agentes(cs) \wedge$
$((\forall n : \text{nombre}) s.hippies.definido(n) \Rightarrow_{\text{L}} s.hippies.obtener(n) =_{\text{obs}} posEstYHippie(n, cs) \vee$
$(\forall n : \text{nombre}) s.estudiantes.definido(n) \Rightarrow_{\text{L}} s.estudiantes.obtener(n) =_{\text{obs}} posEstYHippie(n, cs))$
$(\forall pl : \text{placa}) s.agentes.definido(pl) \Rightarrow_{\text{L}} s.estudiantes.obtener(pl).pos =_{\text{obs}} posAgente(pl, cs))$
$(\forall pl : \text{placa}) s.agentes.definido(pl) \Rightarrow_{\text{L}} s.estudiantes.obtener(pl).cantSanciones =_{\text{obs}} cantSanciones(pl, cs))$
$(\forall pl : \text{placa}) s.agentes.definido(pl) \Rightarrow_{\text{L}} s.estudiantes.obtener(pl).cantCapturas =_{\text{obs}} cantCapturas(pl, cs))$

4.3 Algoritmos

TABLAS(**in** b : `estr`) $\longrightarrow$ *res* : `ConjTrie(string)`
  res $\leftarrow$ *b.tablas.DiccClaves* — O(1)

O(1)

HAYJOIN?(**in** *t1* : `string`, *in t2* : `string`, *in b* : `estr`) $\longrightarrow$ *res* : `bool`
  res $\leftarrow$ Pertenece?(b.Joins, $\langle t1, t2\rangle$)$\vee$Pertenece?(b.Joins, $\langle t2, t1\rangle$) — O(1)

O(1)

CAMPOJOIN(**in** *t1* : `string`, *in t2* : `string`, *in b* : `estr`) $\longrightarrow$ *res* : `campo`
  if Pertenece?(b.Joins, $\langle t1, t2\rangle$) **then** — O(1)
    res $\leftarrow$ Obtener(e.Joins, $\langle t1, t2\rangle$).campoJ — O(1)
  else
    res $\leftarrow$ Obtener(e.Joins, $\langle t2, t1\rangle$).campoJ — O(1)
  end if

O(1)

NUEVADB() $\longrightarrow$ *res* : `estr)`
  res $\leftarrow \langle vacio(), vacio()\rangle$ — O(1)

O(1)

AGREGARTABLA(**in** t : `tabla`, *in b* : `estr`)
  info_tabla $\leftarrow \langle t.cantidadDeAccesos, t\rangle$ — O(1)
  Definir(b.tablas, nombre(t), info_tabla) — O(1)
  Falta hacer algo?

O(1)

INSERTARENTRADA(**in** *reg* : `registro`, *in t* : `string`, *in b* : `estr`)
  T_actual $\leftarrow$ Obtener(b.tablas, t).TActual — O(1)
  agregarRegistro(reg, T_actual) — O(1)
  Falta hacer algo?

O(1)

BORRAR(**in** *cr* : `registro`, *in t* : `string`, *in b* : `estr`)
  T_actual $\leftarrow$ Obtener(b.tablas, t).TActual — O(1)
  borrarRegistro(r, T_actual) — O(1)
  Falta hacer algo?

O(1)

GENERARVISTAJOIN(**in** *t1* : `string`, *in t2* : `string`, *in c* : `campo`, *in/out b* : `estr`)
  Join $\leftarrow$ vacio()
  T_actual1 $\leftarrow$ Obtener(b.tablas, t1).Tactual — O(1)
  T_actual2 $\leftarrow$ Obtener(b.tablas, t2).Tactual — O(1)
  if Definido?(T_actual1.Indices, c) $\wedge$ Definido?(T_actual2.Indices, c) **then**
    ind1 $\leftarrow$ Obtener(T_actual1.Indices, c)
    ind2 $\leftarrow$ Obtener(T_actual2.Indices, c)
    if tipoCampo(T_actual1, c) **then**
      itvalores $\leftarrow$ CrearItConjNat(ind1.PorNat.DiccClaves)
      while HaySiguiente(itvalores) **do**

```
            r1 ← Obtener(ind1.PorNat, Siguiente(itvalores))
            r2 ← Obtener(ind2.PorNat, Siguiente(itvalores))
            cj1 ← AgregarRapido(vacio(), r1)
            cj2 ← AgregarRapido(vacio(), r2)
            nuevor ← combinarRegistros(c, cj1, cj2)
            AgregarRapido(Join,DameUno(nuevor))
            Avanzar(itvalores)
        end while
    else
        itvalores ← CrearItConjString(ind1.PorString.DiccClaves)
        while HaySiguiente(itvalores) do
            r1 ← Obtener(ind1.PorString, Siguiente(itvalores))
            r2 ← Obtener(ind2.PorString, Siguiente(itvalores))
            cj1 ← AgregarRapido(vacio(), r1)
            cj2 ← AgregarRapido(vacio(), r2)
            nuevor ← combinarRegistros(c, cj1, cj2)
            AgregarRapido(Join, DameUno(nuevor))
            Avanzar(itvalores)
        end while
    end if
else
    cjr1 ← T_actual1.registros
    cjr2 ← T_actual1.registros
    Join←combinarRegistros(c, cjr1,cjr2)
end if
info_join ← ⟨0, vacio(), vacio(), c, tipoCampo(T_actual1, c), Join⟩
Definir(b.Joins, ⟨t1, t2⟩ , info_join)
```

O(1)

```
BORRARJOIN(in t1 : string, in t2 : string, in/out b : estr)
if Pertenece?(b.Joins, ⟨t1, t2⟩ ) then
    Borrar(b.Joins, ⟨t1, t2⟩ )
else
    if Pertenece?(b.Joins, ⟨t2, t1⟩ ) then
        Borrar(b.Joins, ⟨t2, t1⟩)
    end if
end if
```

O(1)