Algoritmos y Estructuras de Datos II

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

Trabajo Pràctico 1

Integrante	LU	Correo electrónico
Gustavo Alfonsin	117/17	gustavoalfonsin@gmail.com
Walter Ariel Baya	368/18	walterbaya@yahoo.com
Manuel Avalos	82/18	manuel.avalos.98@gmail.com
Franco Colombini	341/18	francocolombini2013@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. TAD BASE DE DATOS

p

TAD BASE DE DATOS

igualdad observacional

$(\forall b, b' : \text{bd})\ (b =_{\text{obs}} b' \iff (\text{tablas(b)} =_{\text{obs}} \text{tablas(b')}))$

parámetros formales

géneros tabla, NombreTabla

géneros bd

exporta bd, observadores, generadores, borrarTabla, registrar, borrarRegistro, Resolver

usa Valor, NombreTabla,NombreCampo, Registro, Tabla, Consulta, Conjunto(Registro), Diccionario(NombreTabla, tabla)

observadores básicos

tablas	: bd b	$\longrightarrow$ dicc(nombre_tabla,tabla)

generadores

crear	:	$\longrightarrow$ bd
AgTabla	: bd b $\times$ tabla t $\times$ NombreTabla nt	$\longrightarrow$ bd $\{\neg\ \text{def?(nt,tablas(b))} \wedge \emptyset\text{?(registros(t))}\}$
borrar	: bd $\times$ nombre_tabla	$\longrightarrow$ bd
registrar	: bd b $\times$ nombre_tabla nt $\times$ registro r	$\longrightarrow$ bd $\{\text{def?(nt,tablas(b))} \wedge_{\text{L}} \text{campos(r)} =_{\text{obs}} \text{campos(obtener(nt,tablas(b)))}\}$

otras operaciones

borrarTabla	: nombre_tabla nt $\times$ Dicc(nombre_tabla $\times$ tabla)	$\longrightarrow$ dicc(nombre_tabla, tabla)
borrarRegistro	: bd b $\times$ nombre_tabla nt $\times$ valor	$\longrightarrow$ bd $\{\text{def?(nt,tablas(b))} \wedge_{\text{L}} \text{v} =_{\text{obs}} \text{clave(obtener(nt, tablas(b)))}\}$
TablaCompleta	: bd b $\times$ consulta c	$\longrightarrow$ conj(registro) $\{\text{tipo\_consulta(c)} \in \{FROM\}\}$
Filtrar	: bd b $\times$ consulta c	$\longrightarrow$ conj(registro) $\{\text{tipo\_consulta(c)} \in \{SELECT\}\}$
FiltrarConj	: conj(registro) rs $\times$ nombre_campo $\times$ valor	$\longrightarrow$ conj(registro)
Renombre	: bd $\times$ consulta c	$\longrightarrow$ conj(registro) $\{\text{tipo\_consulta(c)} \in \{RENAME\}\}$
RenombreConj	: conj(registro) $\times$ nombre_campo $\times$ nombre_campo	$\longrightarrow$ conj(registro)
MatchCampo	: bd $\times$ consulta c	$\longrightarrow$ conj(registro) $\{\text{tipo\_consulta(c)} \in \{MATCH\}\}$
MatchCampoConj	: conj(registro) $\times$ nombre_campo $\times$ nombre_campo	$\longrightarrow$ conj(registro)
Proyección	: bd $\times$ consulta c	$\longrightarrow$ conj(registro) $\{\text{tipo\_consulta(c)} \in \{PROJ\}\}$
ProyecciónConj	: conj(registro) $\times$ conj(nombre_campo)	$\longrightarrow$ conj(registro)
ProyecciónReg	: conj(nombre_campo) $\times$ registro)	$\longrightarrow$ registro
Intersección	: bd $\times$ consulta	$\longrightarrow$ conj(registro) $\{\text{tipo\_consulta(c)} \in \{INTER\}\}$
Unión	: bd $\times$ consulta	$\longrightarrow$ conj(registro)

			{tipo_consulta(c) ∈ {$UNION$}}
ProductoCartesiano	: bd × consulta c	⟶	conj(registros) {tipo_consulta(c) ∈ {$PRODUCT$}}
ProductoConj	: conj(registro) × conj(registro)	⟶	conj(registro)
multiplicar	: registro × conj(registro)	⟶	conj(regsitro)
combinar	: registro $r1$ × registro $r2$ × conj(nombre_campo) cn	⟶	registro {cn ⊆ campos(r2)}
Resolver	: bd × consulta	⟶	conj(registro)

axiomas

tablas(crear)	≡	vacio
tablas(AgTabla(b,t,nt))	≡	definir(nt,t,tablas(b))
tablas(borrar(b, nt))	≡	borrarTabla(nt, tablas(b))
tablas(registrar(b,nt,r))	≡	definir(nt,insertar(obtener(nt,tablas(b)),r),tablas(b))
borrarRegistro(b,nt,v)	≡	AgTabla(borrar(b,nt), borrar(obtener(b,nt),v),nt)
TablaCompleta(b,c)	≡	**if** def?(nombre_tabla(c),tablas(b)) **then** registros(obtener(nombre_tabla(c),b)) **else** ∅ **fi**
Filtrar(b,c)	≡	FiltrarConj(resolver(subconsulta$_1$(c),b), campo$_1$(c),valor(c))
FiltrarConj(rs,nc,v)	≡	**if** ∅?(rs) **then** ∅ **else** **if** (nc ∈ campos(dameUno(rs)) $\wedge_L$ dameUno(rs)[nc] = v) **then** Ag(dameUno(rs),FiltrarConj(sinUno(rs),nc,v)) **else** FiltrarConj(sinUno(rs),nc,v) **fi** **fi**
Renombre(b,c)	≡	RenombrarConj(resolver(subconsulta$_1$,b),campo$_1$(c),campo$_2$(c))
RenombreConj(rs,nc$_1$,nc$_2$)	≡	**if** ∅?(rs) **then** ∅ **else** Ag(renombrar(dameUno(rs),nc$_1$,nc$_2$),RenombreConj(sinUno(rs))) **fi**
MatchCampo(b,c)	≡	MatchCampoConj(resolver(subconsulta$_1$(c),b),campo$_1$,campo$_2$)
MatchCampoConj(rs,c$_1$,c$_2$)	≡	**if** c$_1$ ∈ campos(dameUno(rs)) ∧ c$_2$ ∈ campos(dameUno(rs)) $\wedge_L$ dameUno(rs)[c$_1$] = dameUno(rs)[c$_2$] **then** Ag(dameUno(rs),MatchCampoConj(sinUno(rs),c$_1$,c$_2$)) **else** MatchCampoConj(sinUno(rs),c$_1$,c$_2$) **fi**
Proyección(b,c)	≡	ProyecciónConj(resolver(subconsulta$_1$(c),b),conjunto_campos(c))
ProyecciónConj(rs,cnc)	≡	**if** ∅?(rs) **then** ∅ **else** Ag(ProyecciónReg(dameUno(rs),cnc),ProyecciónConj(sinUno(rs),cnc)) **fi**

ProyecciónReg(r,cnc) ≡ **if** ∅?(cnc) **then**
r
else
ProyecciónReg(borrarCampo(r,dameUno(cnc)),sinUno(cnc))
fi

Intersección(b,c) ≡ Resolver(b,subconsulta$_1$(c)) ∩ Resolver(b,subconsulta$_2$(c))

Unión(b,c) ≡ Resolver(b,subconsulta$_1$(c)) ∪ Resolver(b,subconsulta$_2$(c))

ProductoCartesiano(b,c) ≡ productoConj(Resolver(b,subconsulta$_1$(c)),Resolver(b,subconsulta$_2$(c)))

ProductoConj(rs$_1$,rs$_2$) ≡ **if** ∅?(rs$_1$) ∨ ∅?(rs$_2$) **then**
∅
else
multiplicar(dameUno(rs$_1$),rs$_2$) ∪ ProductoConj(sinUno(rs$_1$),rs$_2$)
fi

multiplicar(r,rs) ≡ **if** ∅?(rs) **then**
∅
else
Ag(combinar(r,dameUno(rs),campos(dameUno(rs))),multiplicar(r,sinUno(rs)))
fi

combinar(r$_1$,r$_2$,cnc) ≡ **if** ∅?(cnc) **then**
r$_1$
else
if dameUno(cnc) ∈ campos(r$_1$) **then**
combinar(r$_1$,r$_2$,sinUno(cnc))
else
definir(combinar(r$_1$,r$_2$,sinUno(cnc),dameUno(cnc),r$_2$[dameUno(cnc)])
fi
fi

Resolver(b,c) ≡ **if** tipo_consulta = FROM **then**
TablaCompleta(b,c)
else
if tipo_consulta = SELECT **then**
Filtrar(b,c)
else
if tipo_consulta = MATCH **then**
MatchCampo(b,c)
else
if tipo_consulta = PROJ **then**
Proyección(b,c)
else
if tipo_consulta = RENAME **then**
Renombre(b,c)
else
if tipo_consulta = INTER **then**
Intersección(b,c)
else
if tipo_consulta = UNION **then**
Unión(b,c)
else
ProductoCartesiano(b,c)
fi
fi
fi
fi
fi
fi
fi

Fin TAD

2. Extensión Tads

TAD REGISTRO

otras operaciones

renombrar : registro r × nombre_campo nc_1 × nombre_campo nc_2 ⟶ registro

borrarCampo : registro r × nombre_campo nc ⟶ registro

axiomas

renombrar(nuevo,nc_1,nc_2) ≡ nuevo

renombrar(definir(r,c,v),nc_1,nc_2) ≡ **if** c = nc_1 **then**
    definir(renombrar(r),nc_2,v)
else
    definir(renombrar(r),c,v)
fi

borrarCampo(nuevo,nc) ≡ nuevo

borrarCampo(definir(r,c,v)) ≡ **if** c ≠ nc **then** definir(borrarCampo(r),c,v) **else** borrarCampo(r) **fi**

Fin TAD

3. Módulo Base de datos

El modulo Base de datos provee una manera de construir bases de datos en las que se puede agregar tablas con un nombre asociado único (solo se permite agregar tablas si su nombre no está definido), registrar datos en cada tabla mediante registros y borrar tablas y registros particulares. Las tablas de una base de datos pueden ser observadas mediante un diccionario donde las claves son los nombres y los valores son las tablas. Ademas, el módulo permite relizar diferentes consultas sobre sus tablas para extraer información como un conjunto de registros.

Interfaz

se explica con: BASE DE DATOS

géneros: `bd`

3.1. Operaciones

TABLAS(in b : bd) → *res* : `dicc(nombre_tabla,tabla)`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ tablas(b)}
Descripción: devuelve un diccionario con cada nombre de tabla con una tabla asociada como valor

CREAR() → *res* : `baseDeDatos`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ crear}
Descripción: Crea una base de datos

AGTABLA(**in/out** *bd*: `baseDeDatos`, in t : tabla, in Nt : nombre_tabla)
Pre ≡ {¬ def?(Nt,tablas(b)) ∧ ∅?(registros(t) ∧ bd =$_{obs}$ bd$_0$}
Post ≡ {bd =$_{obs}$ AgTabla(bd$_0$,t,Nt)}
Descripción: Introduce una nueva tabla a una base de datos

BORRAR(**in/out** *b*: `bd`, in nt : nombre_tabla)
Pre ≡ {b = b$_0$}
Post ≡ {res =$_{obs}$ borrar(b$_0$,nt)}
Descripción: si esta definida, borra la tabla con el nombre nt, si no esta definida no hace nada

REGISTRAR(**in/out** *b*: `bd`, in nt : nombre_tabla, r : registro)
Pre ≡ {b = b$_0$ ∧ def?(nt,tablas(b)) ∧$_L$ campos(r) =$_{obs}$ campos(obtener(nt,tablas(b)))}
Post ≡ {b =$_{obs}$ registrar(b$_0$,nt,r)}
Descripción: registra en la tabla correspondiente a nt el registro r

RESOLVER(in b : bd, in cons : consulta) → *res* : `conj(registro)`
Pre ≡ {*true*}
Post ≡ {res =$_{obs}$ resolver(b,cons)}
Descripción: Dada una base de datos b y una consulta cons devuelve el conjunto de registros resultante de efectuar la consulta dada en dicha base

Representación

La base de datos se representa con un diccionario implementado sobre tries. Las claves son los nombres de las tablas y los significados las tablas correspondientes.

`bd` **se representa con** `base`

diccTrie(nombreTabla, tabla)

Rep : base ⟶ bool
Rep(*b*) ≡ true ⟺ true

Abs : base b $\longrightarrow$ dicc(nt,tabla) {Rep(b)}

Abs(b) $\equiv$ c / c $=_{\text{obs}}$ tablas(b)

Algoritmos

---

iCrear() $\rightarrow$ res : base

res $\leftarrow$ vacío()

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

Justificación: por ser la operación definir de un diccTries

---

iAgTabla(**in** b : `base`, **in** nt : `String`,**in** t : `a` bla)

b.definir(nt,tabla)

Complejidad: $\mathcal{O}(|t| + n(|c| + |v|))$ donde n es la cantidad de registros que tiene la tabla y |c| y |v| el nombre de campo mas largo y v el valor mas largo de los registros de la tabla.

---

iBorrar(**in** b : `base`, **in** nt : `String`)

b.borrar(nt)

Complejidad: $\mathcal{O}(|t|)$

Justificación: por ser la operación borrar de un diccTries

---

iRegistrar(**in/out** b : `base`,**in** nt : `String` **in** r : `registro`

b.obtener(nt).insertar(r)

Complejidad: $\mathcal{O}(|t| + |v|)$

Justificación: Obtener la tabla de la base a partir de su nombre cuesta $\mathcal{O}(|t|)$ y luego Insertar cuesta $\mathcal{O}(|v|)$

---

iCombinar(**in** $t1$: `tabla`, **in** $t2$: `tabla`)

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

Justificación: por ser la operación borrar de un diccTries

---

iTablaCompleta(**in** b : `base`, **in** nt : `String`) $\rightarrow$ res : conj(reg)

res $\leftarrow$ Vacío()

tabla $\leftarrow$ base.obtener(nt)

for r **in** registros(tabla) **do**

    res.AgregarRapido(r)

end for

Complejidad: $\mathcal{O}(|t| + n(|c| + |v|))$ donde |c| y |v| es el costo de copy(c) y copy(v)

Justificación: Debido a que obtener la tabla tiene costo $\mathcal{O}(|t|)$ y luego copiarla es $\mathcal{O}(n(|c|+|v|))$ ya que es un diccTrie y como tiene n valores ya que la cantidad de valores es igual a la cantidad de registros en la tabla y finalmente el agregarRapido de conjunto lineal tiene complejidad constante pero como lo hago la misma cantidad que la cantidad de registros, la complejidad total será $\mathcal{O}(|t| + n(|c| + |v|))$

---

iSelectConClave(**in** t : `tabla`,**in** c : `clave` **in** *valor* : `String`) → res : conj(Registro)

res ← t.registros.obtener(valor)
Complejidad: $\mathcal{O}(|t| + |v| + |c|)$
Justificación: Como el campo por el que estoy filtrando es el campo clave de la tabla, se que el registro es unico. Entonces encuentro la tabla en |t|, luego comparo el campo dado por parametro con el campo clave de la tabla en |c|, despues busco el registro en la tabla en |v| y finalmente copiar este diccionario tendra complejidad de peor caso $\mathcal{O}(n(|c| + |v|))$ pero sabemos que n es 1, por lo tanto queda $\mathcal{O}(|t| + 2|v| + 2|c|) = \mathcal{O}(|t| + |v| + |c|)$

---

iSelectSinClave(**in** t : `tabla`,**in** c : `String` **in** *valor* : `String`) → res : conj(reg)

```
res ← Vacío
indiceCampo ← tabla.indices.obtener(c)
for reg r in registros(tabla) do
    if r.obtenerValor[indiceCampo] == valor then
        res.AgregarRapido(r)
    end if
end for
```

Complejidad: $\mathcal{O}(|t| + |c| + n|v| + k(|c|, |v|))$
Justificación: Busco la tabla en |t|, me fijo si el campo dado por parametro coincide con la clave de la tabla en |c| y ademas busco el indice correspondiente en el diccTrie indices en |c| = 2|c|, luego para cada uno de los registros en la tabla comparo el valor del registro con el valor por parametro n|v| (acceder a el valor en el registro es $\mathcal{O}$ dado que tengo un vector con valores y el indice donde se encuentra), finalmente copio todos los que coincidan k(copy(c) + copy(v)) = k(|c| + |v|).

---

iJoinConClaves(**in** c : `consulta`,**in** c : `campo1` **in** c : `campo2`) → res : conj(Registro)

```
res ← Vacio
Puntero(masChica) ← min(c.subconsulta1.subconsulta1.tabla, c.subconsulta1.subconsulta2.tabla)
Puntero(masGrande) ← max(c.subconsulta1.subconsulta1.tabla, c.subconsulta1.subconsulta2.tabla)
for v in masChica.claves do
    if def?(v,max(c.masGrande.registros) then
        AgregarRapido(res,combinar(obtener(v,masChica),obtener(v,masGrande)))
    end if
end for
```

Complejidad:$\mathcal{O}(min(t1, t2)|v|)$
Justificación: Tiene que revisar todas las claves de la tabla con menor cantidad de registros y cada vez que pasa por el if cuesta $\mathcal{O}(|v|)$ luego el agregarRapido y el combinar cuestan $\mathcal{O}(|1|)$ y todo esto en definitiva cuesta $\mathcal{O}(min(t1, t2)|v|)$

---

iSelectConClaveDeSelectSinClave(**in** t : `tabla`,**in** $c1$: `clave` **in** *valor*1 : `String`,**in** $c2$: `campo` **in** *valor*2 : `String`) → res : (Registro)

```
Miregistro = tabla.registros.obtener(valor1)
res ← Vacío()
if Miregistro.obtener(c2)==v2 then
    res.Agregar(Miregistro)
end if
```

Complejidad: $\mathcal{O}(|v| + |c|)$
Justificación: Al ser registros un dicc de trie [valor,registro] obtengo el unico registro que coincide con el valor de la clave c1 en $\mathcal{O}(|v|)$,luego comparo si el campo c2 del registro coincide con el valor v2 en $\mathcal{O}(|c|)$.

---

iInterseccionDosSelecciones(**in** $t1$: `tabla`,**in** $t2$: `tabla`, **in** $c1$: `String`,**in** $c2$: `String` **in** $valor1$: `String`, **in** $valor2$: `String`) → res : conj(reg)

```
res ← Vacío
indiceCampo1 ← tabla.indices.obtener(c1)
indiceCampo2 ← tabla.indices.obtener(c2)
for reg r in registros(tabla) do
    if r.obtenerValor[indiceCampo1] == valor1 ∧ r.obtenerValor[indiceCampo2] == valor2 then
        res.AgregarRapido(r)
    end if
end for
```

Complejidad: $\mathcal{O}(|t| + |c| + n|v| + k(|v| + |c|))$

Justificación: La complejidad se da de manera similar al select sin clave. Busco las tabla en 2|t|, chequeo si los campos son clave en 2|c| y obtengo el indice correspondiente con el campo en 2|c| (queda 4|c|), para cada uno de los registros me fijo si el valor coincide(teniendo el indice accedo al vector en O(1)) en |v| (comparacion), y finalmente copiar los k registros que cumplan cuesta k(|v| + |c|). Como las constantes las puedo descartar me queda la complejidad requerida.

---

iHacerSelect(**in** b: `bd`, **in** $cons$: `consulta`) → res : conj(Registro)

```
sub11 ← cons.subconsulta1.subconsulta1
sub12 ← cons.subconsulta1.subconsulta2
sub1 ← cons.subconsulta1
if sub1 == FROM ∧_L def?(cons.campo1, sub1.tabla.registros) then
    if cons.campo1==sub1.tabla.campoClave then
        res ← SelectConClave(sub1.tabla,cons.campo1,cons.valor)
    else
        res ← SelectSinClave(sub1.tabla, const.campo1, cons.valor)
    end if
else
    sub11 ← cons.subconsulta1.subconsulta1
    if sub1 == SELECT ∧ sub11 == FROM ∧ sub1.campo1 == sub1.tabla.campoClave ∧ sub11.campo1 ! =
sub1.tabla.campoClave then
        res ← selectConClaveDeSelectSinClave(sub11.tabla, sub1.campo1, sub1.valor1,sub11.campo1, sub11.valor1)
    else
        res ← Resolver(b, cons.subconsulta1)
        for r in res do
            if r.Obtener(cons.campo1) ! = valor then
                res.Borrar(r)
            end if
        end for
    end if
end if
```

iHacerMatch(**in** b : `bd`, **in** *cons* : `consulta`) → res : conj(reg)

```
sub11 ← cons.subconsulta1.subconsulta1
sub12 ← cons.subconsulta1.subconsulta2
c1 ← def?(cons.campo2,obtener(dameUno(claves(sub12.tabla.registros)),(sub12.tabla.registros)))
c2 ← def?(cons.campo1,obtener(dameUno(claves(sub11.tabla.registros)),(sub11.tabla.registros))
if cons.subconsulta1.tipo == PRODUCT ∧ sub11 == FROM ∧ sub12 == FROM ∧ c1 ∧ c2  then
    res ← joinConClaves(cons.subconsulta1,cons.campo1,cons.campo2)
else
    res ← resolver(b,cons.subconsulta1)
    for r in res do
        if obtener(cons.campo1,res) ! = obtener(cons.campo2,res) then
            borrar(r,res)
        end if
    end for
end if
```

iHacerProj(**in** b : `bd`, **in** *cons* : `consulta`) → res : conj(reg)

```
definidos ← true;
res ← Resolver(b,cons.subconsulta1)
unRegistro ← res.dameUno
for c in cons.conj_Campos do
    if unRegistro.campos.pertenece(c) ∧ definidos then
    else
        definido ← false
    end if
end for
if definidos ∧ res.Cardinal>0 then
    for r in res do
        for c in cons.conj_Campos do
            borrarCampo(r,c)
        end for
    end for
else
    res ← Vacio
end if
```

iHacerInter(**in** b : `bd`, **in** *cons* : `consulta`) → res : conj(reg)

```
if cons.subconsulta1.tipo==SELECT ∧ cons.subconsulta2.tipo==SELECT ∧
cons.subconsulta1.subconsulta1.tipo==FROM ∧
cons.subconsulta2.subconsulta1.tipo==FROM ∧
cons.subconsulta2.subconsulta1.nombreTabla == cons.subconsulta1.subconsulta1.nombreTabla then
    t1 ← b.obtener(cons.subconsulta1.subconsulta1.nombreTabla)
    t2 ← b.obtener(cons.subconsulta2.subconsulta1.nombreTabla)
    valor1 ← cons.subconsulta1.valor
    valor2 ← cons.subconsulta2.valor
    campo1 ← cons.subconsulta1.campo1
    campo2 ← cons.subconsulta2.campo1
    res ← iInterseccionDosSelecciones(t1,t2,campo1,campo2,valor1,valor2);
else
    res ← Vacio
    res1 ← Resolver(b,cons.subconsulta1)
    res2 ← Resolver(b,cons.subconsulta2)
    for r in res1 do
        if Pertenece(r,res2) then
            Agregar(r,res)
        end if
    end for
end if
```

iResolver(**in** b : `bd`, **in** *cons* : `consulta`) → res : conj(registro)

```
if cons.tipo == FROM then
    res ← tablaCompleta(b,cons.tabla)
else if cons.tipo == SELECT then res ← HacerSelect(b,cons)
else if cons.tipo == MATCH then res ← HacerMatch(b,cons)
else if cons.tipo == PROJ then res ← HacerProj(b,cons)
else if cons.tipo==RENAME then
    res ← Resolver(b,cons.subconsulta1)
    for r in res do
        renombrar(r,cons.campo1,cons.campo2)
    end for
else if cons.tipo == INTER then
    ← HacerInter(b,cons)
else if cons.tipo == UNION then
    res ← Resolver(cons.subconsulta1)
    res2 ← Resolver(cons.subconsulta2)
    for r in res do
        if !Pertenece(r,res2) then
            Agregar(r,res)
        end if
    end for
end if
```

Complejidad: $\mathcal{O}(costoFuncion)$.

Justificación: donde costo funcion depende del tipo de consulta que se esté resolviendo, la cual en general depende de los costos de resolver consultas anteriores(comportamiento recursivo), salvo casos especificos(casos base).

4. Módulo Registro

El módulo registro provee una manera de almacenar valores(strings) con un campo asociado. El módulo permite crear nuevos registros, definir nuevos campos con sus respectivos valores, renombrar campos ya definidos y eliminar campos. Se pueden observar: el conjunto de campos y el valor asociado a un campo.

Interfaz

se explica con: REGISTRO

géneros: `registro`

4.1. Operaciones

NUEVO() $\rightarrow$ *res* : `registro`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ nuevo}
Descripción: genera un nuevo registro

DEFINIR(**in/out** *r* : `registro`, in nc : nombre_campo, in v : valor)
Pre $\equiv$ {r = r_0}
Post $\equiv$ {res $=_{obs}$ definir(r_0,nc,v)}
Descripción: define un campo nuevo(si no existe) con su valor o pisa el valor anterior del campo

CAMPOS(in r :registro) $\rightarrow$ *res* : `conj(nombreCampo)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ campos(r)}
Descripción: da el conjunto de campos de un registro

•[•](in r : registro, in nc : nombreCampo) $\rightarrow$ *res* : `valor`
Pre $\equiv$ {nc $\in$ campos(r)}
Post $\equiv$ {res $=_{obs}$ r[nc]}
Descripción: devuelve el valor asociado al campo nc

RENOMBRAR(**in/out** *r* : `registro`, **in** *nc*1 : `nombreCampo`, **in** *nc*2 : `nombreCampo`)
Pre $\equiv$ {r $=_{obs}$ r0}
Post $\equiv$ {r $=_{obs}$ renombrar(r0,nc1,nc2)}
Descripción: devuelve el registro con el campo nc1 cambiado nc2 BORRARCAMPO(**in/out** *r* : `registro`, in nc : nombreCampo)
Pre $\equiv$ {r $=_{obs}$ r0}
Post $\equiv$ {r $=_{obs}$ borrarCampo(r0,nc)}
Descripción: devuelve el registro r sin el campo nc

OBTENERVALOR(**in** *r* : `registro`, **in** *indice* : `Nat`) $\rightarrow$ *res* : `valor`
Pre $\equiv$ {true}
Post $\equiv$ {}
Complejidad: nc $\in$ campos(r) res $igobs r.obtenerValor(indice) [] [devuelve el valor indicado por el indice]

Representación El registro se representa como Tupla(registros: diccTrie(nombreCampo, valor),datos: vector(valor))

Rep : registro $\longrightarrow$ bool

Rep(r) $\equiv$ true $\iff$ Para todo valor en el vector datos tiene que existir un campo en registros de manera que su significado en registros sea dicho valor y viceversa. El tamaño de datos es igual a la cantidad de claves en registros

Abs : registro r $\longrightarrow$ secuencia(tupla(campo,valor)) {Rep(r)}

Abs(r) $\equiv$ c / ($\forall$ n : nombre_campo)(n $\in$ campos(c)$\Longleftrightarrow$ definido(n,r.registros) $\wedge_{\text{L}}$obtener(n,r.registros) $=_{\text{obs}}$ c[n])

Algoritmos

---

iNuevo()$\rightarrow$ res : registro

res $\leftarrow$ Vacía()

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

Justificación: Es la complejidad de crear una lista enlazada.

---

iDefinir(**in/out** r : `registro`, **in** nc : `nombreCampo`, **in** v : `valor`) $\rightarrow$ res : registro

r.agregarAtras(<nc,v>)

Complejidad: $\mathcal{O}(copy(nc) + copy(v))$

Justificación: Agregar un elemento a una lista enlazada es lineal con respecto a la cantidad de elementos, pero considerando la cantidad de campos constante, la complejidad queda unicamente dependiendo del costo de copiar los elementos.

---

iCampos(**in/out** r : `registro`)$\rightarrow$ res : conj(nombreCampo)

```
res← Vacío()
for p in r do
    res.AgregarRapido(p.nombreCampo)
end for
```

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

Justificación: La cantidad de campos es constante por lo tanto la complejidad queda constante.

•[•](**in** r : `registro`, **in** nc : `nombreCampo`) $\rightarrow$ res : registro

```
res ← Nuevo()
for campoValor in r do
    if campoValor.nombreCampo == nc then
        res ← campoValor.valor
    end if
end for
```

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

Justificación: Por ser los campos constantes en numero esto resulta ser de complejidad constante

---

irenombrar(**in/out** r : `reg`, **in** $nc1$: `String`, **in** $nc2$: `String`)

```
for CampoValor in r do
    if CampoValor.nombreCampo == nc1 then
        CampoValor.nombreCampo = nc2
    end if
end for
```

Complejidad: $\Theta(copy(nc2))$

Justificación: Recorrer el registro es constante puesto que el numero de campos es acotado, luego la complejidad dependera de el costo de copiar el nuevo nombre.

---

iborrarCampo(**in**/**out** r : `reg`, **in** nc : `String`)

```
nuevoReg ← r
for CampoValor in r do
    if CampoValor.nombreCampo != nc then
        nuevoReg.AgregarAtras(CampoValor)
    end if
end for
```

Complejidad: $\Theta(copy(r))$

Justificación: En el peor caso el campo nc no pertenece al registro y tengo que copiar el registro entero.

5. Módulo Tabla

El módulo tabla provee un manera de construir tablas mediante registros. El módulo permite crear tablas nuevas(especificando cuales son los campos requeridos y cual es la clave), insertar y borrar registros(siempre y cuando coincidan con los campos admitidos por la tabla). Se pueden observar sus campos, su campo clave y el conjunto de registros.

Interfaz

usa: `REGISTRO, CONJ(NOMBRECAMPO), NOMBRECAMPO`

se explica con: TABLA

géneros: `tabla`

5.1. Operaciones

CAMPOS(in t : tabla) $\rightarrow$ *res* : `conj(nombre_campo)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ campos(t)}
Descripción: Dada una tabla devuelve el conjunto de nombres de campo de la misma

CLAVE(in t : tabla) $\rightarrow$ *res* : `nombre_campo`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ campos(t)}
Descripción: Dada una tabla devuelve el nombre del campo clave

REGISTROS(in t : tabla) $\rightarrow$ *res* : `conj(nombre_campo)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ registros(t)}
Descripción: Dada una tabla devuelve el conjunto de nombres de campo de la misma

NUEVA(in Cn : conj(nombre_campo), in n : nombre_campo) $\rightarrow$ *res* : `tabla`
Pre $\equiv$ {n $\in$ Cn}
Post $\equiv$ {res $=_{obs}$ nueva(Cn,n)}
Descripción: Genera una nueva tabla

INSERTAR(**in/out** *t*: `tabla`, in r : registro)
Pre $\equiv$ {campos(t) = campos(r) $\wedge$ t $=_{obs}$ t_0}
Post $\equiv$ {t $=_{obs}$ insertar(t,r)}
Descripción: agrega un nuevo registro a una tabla

BORRAR(in/out t : tabla, in v : valor)
Pre $\equiv$ {t $=_{obs}$ t_0}
Post $\equiv$ {t $=_{obs}$ borrar(t_0,v)}
Descripción: Borra un registro, si esta en la tabla, cuyo valor en el campo clave sea igual al valor dado

INSERTARREGISTRO(in k : nombre_campo, in c : registro, inout rs: conj(registro))
Pre $\equiv$ {rs = rs_0 $\wedge$ k $\in$ campos(r) $\wedge$ ($\forall$ l : registro)(l $\in$ rs $\Rightarrow$ k $\in$ campos(l))}
Post $\equiv$ {rs $=_{obs}$ insertarRegistro(k,c,rs_0)}
Descripción: Inserta un registro dentro de un conjunto de registros

BORRARREGISTRO(in k : nombre_campo, in v : valor , inout Rs : conj(registro))
Pre $\equiv$ {rs = rs_0 $\wedge$ ($\forall$ r : registro) (r $\in$ Rs $\Rightarrow$ k $\in$ campos(r))}
Post $\equiv$ {rs $=_{obs}$ borrarRegistro(k,c,rs_0)}
Descripción: Borra un registro dentro de un conjunto de registros

Representación La tabla se representa con una tupla que contiene un diccionario sobre tries con los valores correspondientes al campo clave como claves y con las tablas respectivas como significados, una string que representa

el campoClave, un natural que indica la cantidad de registros en la tabla y el conjunto de nombres de campos.

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

donde `tabla` es `tupla`(*indices*: `diccTrie(campo, indice)` , *campoClave*: `String` , *tamaño*: `Nat` , *Info*: `conjunto(registro)` , *registros*: `diccTrie(valorCampoClave,registro)` , *campos*: `conj(string)`)

Rep : tabla $\longrightarrow$ bool

Rep(t) $\equiv$ true $\iff$ campoClave esta en claves de indices y en campos
Campos debe ser igual a las claves de indices
El tamaño es igual a la cantidad de claves de registros
Todo los registros tienen igual cantidad de campos
Todo campo que esta definido en indice esta definido en registros de todos los registros y viceversa
Para todo par de campos pasa que sus indices son distintos
Para toda clave en registros su significado esta en info y viceversa

Abs : tabla t $\longrightarrow$ tabla {Rep(t)}

Abs(t) $\equiv$ s /(campos(t) $=_{\text{obs}}$ s.campos $\wedge_{\text{L}}$ clave(t) $=_{\text{obs}}$ s.campoClave $\wedge_{\text{L}}$ #registros(t) $=_{\text{obs}}$ s.tamaño $\wedge_{\text{L}}$ ($\forall$ v : valor)(v $\in$ claves(s.registros) $\Rightarrow$ ($\exists$ r : registro)(r $\in$ registros(t))$\wedge$ (r $=_{\text{obs}}$ obtener(v, s.registros))

Algoritmos

---

campos(**in** t : `tabla`) $\rightarrow$ res : &conj(String)
res $\leftarrow$ t.campos
Complejidad: $\Theta(1)$
Justificación: Acceder a un elemento de la tupla es constante y se devuelve una referencia al conjunto.

---

clave(**in** t : `tabla`) $\rightarrow$ res : &String
res $\leftarrow$ t.campoClave
Complejidad: $\Theta(1)$
Justificación: Acceder a un elemento de la tupla es constante y se devuelve una referencia al string.

---

iRegistros(**in** t : `tabla`) $\rightarrow$ res : conj(registros)
res $\leftarrow$ t.Info
complejidad:$\mathcal{O}(1)$
justificación: Acceder a un elemento de la tupla es constante, y se devuelve una referencia al conjunto.

---

iNueva(**in** cn : `conj(nombreCampo)`, **in** n : `nombreCampo`) $\rightarrow$ res : tabla
res.registros $\leftarrow$ CrearDiccTries()
res.indice $\leftarrow$ CrearDiccTries()
res.Info $\leftarrow$ Vacio()
res.campoClave $\leftarrow$ n
res.tamaño $\leftarrow$ 0
res.campos $\leftarrow$ cn
complejidad: $\Theta(|1|)$
justificación: Por ser la operación de definir de un diccTries y las asignaciones tienen costo constante.

---

iInsertar(**in/out** t : `tabla`, **in** r : `registro`)

definir(r[t.campoClave],r,registros)
t.tamaño ← t.tamaño + 1
complejidad: $\Theta(|V|)$
justificación: por ser la operación definir de un diccTries.

---

iBorrar(**in/out** t : `tabla`,**in** v : `valor`)

if definido?(v,t.registros) **then**
    borrar(v,t.registros)
    t.tamaño ← t.tamaño - 1
    eliminar(r,t.Info)
end if
complejidad: $\Theta(|V|)$
justificación: por hacer uso de la operacion borrar de diccTries

---

iInsertarRegistro(**in/out** t : `tabla`,**in** k : `nombre_`*campo*,**in** r : `registro`)

t.registros.definir(r.(t.campoClave),r)
complejidad: $\Theta(|v|+)$
justificación: Es la complejidad de agregar un elemento a un conjunto

---

iBorrarRegistro(**in/out** t : `tabla`,**in** k : `nombre_campo`,**in** v : `valor`)

rt = Vacio
it = true
for c **in** **do**
    if (v**in** C : `l` aves(dameUno(rs) **then**
        rs = rs - dameUno(rs)
        it = False
    else
        rt = rt + dameUno(rs)
        rs= rs -dameuno(rs)
    end if
end for
rs = rs + rt
complejidad: $\Theta(n.|r|)$
justificación: Es la complejidad de recorrer el conjunto buscando el registro y para cada registro ver si el valor pertenece a las claves

---

6. Módulo Consulta

El modulo consulta provee un manera de escribir "preguntas"para hacerle a una base de datos, es decir, especificar que información se quiere extraer de ella. Para ello permite crear diferentes tipos de consultas a partir de otras consultas, a exepción de FROM que toma el nombre de una tabla.

Interfaz

usa: `REGISTRO, CONJ(NOMBRECAMPO), NOMBRECAMPO`

se explica con: CONSULTA

géneros: `consulta`

6.1. Operaciones

FROM(in nt : nombre_tabla) → *res* : `consulta`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ FROM(nt)}
Descripción: devuelve la consulta FROM sobre el nombre de tabla nt

SELECT(in cons : consulta, in c : nombre_campo, in v : Valor) → *res* : `consulta`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ SELECT(cons,c,v)}
Descripción: devuelve la consulta SELECT sobre la consulta cons, el campo c y el valor v

MATCH(in cons : consulta, in nc_1 : nombre_campo, in nc_2 : nombre_campo) → *res* : `consulta`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ MATCH(cons,nc_1, nc_2)}
Descripción: devuelve la consulta MATCH sobre la consulta cons y los campos nc_1 y nc_2

PROJ(in cons : consulta, in : conj(nombre_campo), in nc : nombre_campo) → *res* : `consulta`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ PROJ(cons,nc)}
Descripción: devuelve la consulta PROJ sobre la consulta cons y los campos nc

RENAME(in cons : consulta, in nc_1 : nombre_campo, in nc_2 : nombre_campo) → *res* : `consulta`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ RENAME(cons,nc_1, nc_2)}
Descripción: devuelve la consulta RENAME sobre la consulta cons y los campos nc_1 y nc_2

INTER(in *cons*_1 : consulta, in *cons*_2 : consulta) → *res* : `consulta`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ INTER(*cons*_1,*cons*_2)}
Descripción: devuelve la consulta INTER sobre las consulta *cons*_1 y *cons*_2

UNION(in *cons*_1 : consulta, in *cons*_2 : consulta) → *res* : `consulta`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ UNION(*cons*_1,*cons*_2)}
Descripción: devuelve la consulta UNION sobre las consulta *cons*_1 y *cons*_2

PRODUCT(in *cons*_1 : consulta, in *cons*_2 : consulta) → *res* : `consulta`
Pre ≡ {true}
Post ≡ {res =$_{obs}$ PRODUCT(*cons*_1,*cons*_2)}
Descripción: devuelve la consulta PRODUCT sobre las consulta *cons*_1 y *cons*_2

TIPO_CONSULTA(in cons : consulta) → *res* : `tipo_consulta`
Pre ≡ {}
Post ≡ {res =$_{obs}$ tipo_consulta(cons)}
Descripción: devuelve el tipo de consulta

NOMBRE_TABLA(in cons : consulta) $\rightarrow$ res : `nombre_tabla`
Pre $\equiv$ {tipo_consulta(cons) $\in$ $\{FROM\}$}
Post $\equiv$ {res $=_{\text{obs}}$ nombre_tabla(cons)}
Descripción: devuelve el nombre de la tabla que toma la consulta(solo from)

CAMPO$_1$(in cons : consulta) $\rightarrow$ res : `nombre_campo`
Pre $\equiv$ {tipo_consulta(cons) $\in$ $\{SELECT, MATCH, RENAME\}$}
Post $\equiv$ {res $=_{\text{obs}}$ campo$_1$(cons)}
Descripción: devuelve el primer campo que toma como parametro la consulta

CAMPO$_2$(in cons : consulta) $\rightarrow$ res : `nombre_campo`
Pre $\equiv$ {tipo_consulta(cons) $\in$ $\{MATCH, RENAME\}$}
Post $\equiv$ {res $=_{\text{obs}}$ campo$_2$(cons)}
Descripción: devuelve el segundo campo que toma como parametro la consulta

VALOR(in cons : consulta) $\rightarrow$ res : `valor`
Pre $\equiv$ {cons $\in$ $\{SELECT\}$}
Post $\equiv$ {res $=_{\text{obs}}$ valor(cons)}
Descripción: devuelve el valor asociado a una consulta SELECT

CONJ_CAMPOS(in cons : consulta) $\rightarrow$ res : `conj(nombre_campo)`
Pre $\equiv$ {cons $\in$ $\{PROJ\}$}
Post $\equiv$ {res $=_{\text{obs}}$ conj_campos(cons)}
Descripción: devuelve el conjunto de campos asociado a hacer la consulta PROJ

SUBCONSULTA$_1$(in cons : consulta) $\rightarrow$ res : `consulta`
Pre $\equiv$ {cons $\in$ $\{SELECT, MATCH, PROJ, RENAME, INTER, UNION, PRODUCT\}$}
Post $\equiv$ {res $=_{\text{obs}}$ subconsulta$_1$(cons)}
Descripción: devuelve la primer consulta anterior de una consulta

SUBCONSULTA$_2$(in cons : consulta) $\rightarrow$ res : `consulta`
Pre $\equiv$ {cons $\in$ $\{INTER, UNION, PRODUCT\}$}
Post $\equiv$ {res $=_{\text{obs}}$ subconsulta$_2$(cons)}
Descripción: devuelve la segunda consulta anterior de una consulta

Representación Las consultas se representan con una tupla que tiene un string que indica el tipo de consulta y los respectivos parametros que le corresponden a esa consulta en particular(nombreCampo, subconsulta, etc.)

`consulta` se representa con `cons`

donde `cons` es `tupla`(*tipo*: `String` , *tabla*: `String` , *campo1*: `String` , *campo2*: `String` , *valor*: `String` , *conjCampos*: `conj(String)` , *subconsulta1*: `cons` , *subconsulta2*: `cons`)

Rep : consulta $\longrightarrow$ bool

Rep(*cons*) $\equiv$ true $\Longleftrightarrow$

Abs : consulta r $\longrightarrow$ donde `cons` es `tupla`(*tipo*: `String` , *tabla*: `String` , *campo1*: `String` , *campo2*: `String` , *valor*: `String` , *conjCampos*: `conj(String)` , *subconsulta1*: `cons` , *subconsulta2*: `cons`) {Rep(r)}

Abs(r) $\equiv$ c / donde en todos los casos debe pasar que tipoConsulta(c) = cons.tipo

- Si la consulta es FROM cons.tabla = tabla(c)
- Si la consulta es SELECT subconsulta1(c) = cons.subconsulta1 y campo1(c) = cons.campo1 y cons.valor = valor(c)
- Si la consulta es MATCH subconsulta1(c) = cons.subconsulta1 y campo1(c) = cons.campo1 y = campo2(c) = cons.campo2
- Si la consulta es PROJ subconsulta1(c) = cons.subconsulta1 y conjCampos(c) = cons.conjCampos
- Si la consulta es RENAME subconsulta1(c) = cons.subconsulta1 y campo1(c) = cons.campo1 y campo2(c) = cons.campo2
- Si la consulta es INTER subconsulta1(c) = cons.subconsulta1 y subconsulta2(c) = cons.subconsulta2
- Si la consulta es UNION subconsulta1(c) = cons.subconsulta1 y subconsulta2(c) = cons.subconsulta2
- Si la consulta es PRODUCT subconsulta1(c) = cons.subconsulta1 y subconsulta2(c) = cons.subconsulta2

7. Módulo diccionario sobre Trie

El modulo provee un diccionario implementado sobre arboles Trie que permite hacer inserciones, borrados y lecturas con complejidad dependiente, en el peor caso, de la clave mas larga definida.

Interfaz

parámetros formales

géneros κ, σ

función $\bullet = \bullet(\textbf{in } k_1 : \kappa, \textbf{in } k_2 : \kappa) \rightarrow res$: `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} (k_1 = k_2)\}$
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ's

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

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

se explica con: DICCIONARIO(κ, σ)

géneros: `diccTrie`(κ, σ)

7.1. Operaciones básicas de diccionarioTrie

VACÍO() $\rightarrow$ *res* : `diccTrie`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ vacío()}
Complejidad: $\Theta(1)$
Descripción: Crea un diccionario vacío

DEFINIR(**in/out** *d*: `diccTrie`(κ, σ), `in` *k* : κ, `in` *s* : σ)
Pre $\equiv$ {d $=_{\text{obs}}$ d0}
Post $\equiv$ {res $=_{\text{obs}}$ definir(d0,k,s)}
Complejidad: Θ(|k| + copy(k) + copy(s))
Descripción: Define la clave k en el diccionario o pisa el valor anterior de la clave

DEFINIDO?(**in** *d*: `diccTrie`(κ, σ), **in** *k*: κ) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ def?(d,k)}
Complejidad: Θ(|k|)
Descripción: Devuelve true si k esta definido en el diccionario

OBTENER(**in** *d*: `diccTrie`, **in** *k*: κ) $\rightarrow$ *res* : σ
Pre $\equiv$ {def?(d,k)}
Post $\equiv$ {res $=_{\text{obs}}$ obtener(d,k)}
Complejidad: Θ(|k|)
Descripción: Devuelve una referencia al valor asociado a la clave k

BORRAR(**in/out** *d*: `diccTrie`(κ, σ), **in** *k*: κ)
Pre $\equiv$ {d $=_{\text{obs}}$ d0 $\wedge$ def?(d,k)}
Post $\equiv$ {res $=_{\text{obs}}$ borrar(d0, k)}
Complejidad: Θ(|k|)
Descripción: Borra la clave y su significado del diccionario

CLAVES(**in** *d*: `diccTrie`(κ, σ)) $\rightarrow$ *res* : `conj(`κ`)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ claves(d)}
Complejidad: $\Theta(1)$
Descripción: Devuelve el conjunto de claves

8. Decisiones tomadas

- Realizar la consulta FROM sobre una tabla no definida, devuelve vacío.
- Al realizar la consulta SELECT, si hay un registro que no tiene el campo definido, se lo omite.
- Al realizar la consulta MATCH, si hay uno de los dos registros que no tienen el campo correspondiente, se los omite.
- Al realizar RENAME sobre un registro que no tiene el campo a cambiar, devuelve el registro original.