Algoritmos y Estructuras de Datos II

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

Trabajo Práctico 2

"Algebra Relacional"

Integrante	LU	Correo electrónico
Luis Ricardo Bustamante	43/18	luisbustamante097@gmail.com
Leandro Emanuel Rodriguez	521/17	leandro21890000@gmail.com
Ramiro Augusto Ciruzzi	228/17	ramiro.ciruzzi@gmail.com
David Alejandro Venegas	783/18	venegasr.david@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Modulo BaseDeDatos

El modulo BaseDeDatos provee operaciones básicas a realizarse sobre una base de datos. Permite crear una base de datos, agregar o eliminar una tabla, agregar o eliminar registros a una tabla, y también permite ejecutar consultas para obtener registros de una tabla.

Interfaz

se explica con: BaseDeDatos

usa: Registro, Tabla, Consulta, Bool, Nat, ConjuntoLineal

género: `base_de_datos`

NuevaBd() $\rightarrow$ *res* : `BaseDeDatos`
Pre $\equiv$ {true}
Post $\equiv$ {res = nuevaBD}
Complejidad: $O(1)$
Descripción: Crea una nueva base de datos.

AgregarTabla(**in/out** b: *BaseDeDatos*, **in** t: *Tabla*, **in** nt: *NombreTabla*)
Pre $\equiv$ {b = b0}
Post $\equiv$ {b = AgregarTabla(b0, t, nt)}
Complejidad: $O(nt + copy(t))$
Descripción: Inserta una tabla a una base de datos. En caso de que la tabla ya existe en la base, la sobreescribe.

BorrarTabla(**in/out** b: *BaseDeDatos*, **in** nt: *NombreTabla*)
Pre $\equiv$ {b = b0}
Post $\equiv$ {(nt $\notin$ nombreTablas(b0) $\wedge$ b=b0) $\vee$ (nt $\in$ nombreTablas(b0) $\wedge$ b=sacarTabla(b0, nt))}
Complejidad: $O(|nt|)$
Descripción: Elimina una tabla de una base de datos. En caso de que la tabla no exista en la base, la operacion queda sin efecto.

AgregarRegistro(**in/out** b: *BaseDeDatos*, **in** nt: *NombreTabla*, **in** r: *Registro*)
Pre $\equiv$ {b = b0 $\wedge$ (campos(tabla(b, nt)) = campos(r)}
Post $\equiv$ {b = AgregarRegistro(b0, tabla(b0, nt), r))}
Complejidad: $O(|nt| + copy(r))$
Descripción: Inserta una registro en una tabla en una base de datos dándole valor a todos (y solamente a) los campos de la tabla. En caso de que el registro ya existe en la base, lo sobreescribe.

BorrarRegistro(**in/out** b: *BaseDeDatos*, **in** nt: *NombreTabla*, **in** v: *Valor*)
Pre $\equiv$ {b = b0 $\wedge$ nt $\in$ nombreTablas(b0)}
Post $\equiv$ {b = sacarRegistro(b0, tabla(b0, nt), v)}
Complejidad: $O(|nt| + k), k = \#registros$
Descripción: Elimina un registro de una tabla en una base de datos. En caso de que el registro no pertenezca a la tabla, esta queda sin efecto

NombreTablas(**in** b: *BaseDeDatos*) $\rightarrow$ *res* : `Conj(nombreTabla)`
Pre $\equiv$ {true}
Post $\equiv$ {res = nombreTablas(b)}
Complejidad: $O(|n| * k)$
Descripción: Devuelve el conjunto de nombres de todas las tablas de la base de datos.

Tabla(**in** b: *BaseDeDatos*, **in** nt: *NombreTabla*) $\rightarrow$ *res* : `tabla`
Pre $\equiv$ {nt $\in$ nombreTablas(b)}
Post $\equiv$ {res = tabla(b, nt)}
Complejidad: $O(|nt|)$
Descripción: Dados un nombre de tabla y una base de datos, devuelve la tabla correspondiente al nombre en dicha base.

EjecutarConsulta(**in** b: *BaseDeDatos*, **in** q: *Consulta*) $\rightarrow$ *res* : `conj(Registro)`
Pre $\equiv$ {
(TipoConsulta(q) $\in$ $\{FROM\}$ $\wedge$ ExisteTablaEnBd(b, NombreTabla(q))) $\wedge_{\text{L}}$
(TipoConsulta(q) $\in$ $\{SELECT\}$ $\wedge$ RegistrosTienenCampo(ejecutarConsulta(b, subconsulta1(q)), campo1(q))) $\wedge_{\text{L}}$
(TipoConsulta(q) $\in$ $\{MATCH\}$ $\wedge$ RegistrosTienenCampo(ejecutarConsulta(b, subconsulta1(q)), campo1(q)) $\wedge$ RegistrosTienenCampo(ejecutarConsulta(b, subconsulta1(q)), campo2(q))) $\wedge_{\text{L}}$
(TipoConsulta(q) $\in$ $\{RENAME\}$ $\wedge$ RegistrosTienenCampo(ejecutarConsulta(b, subconsulta1(q)), campo1(q)) $\wedge$

¬RegistrosTienenCampo(ejecutarConsulta(b, subconsulta1(q)), campo2(q))) $\wedge_{\text{L}}$
(TipoConsulta(q) $\in \{PRODUCT\} \wedge \neg$ HayCamposEnComun(ejecutarConsulta(b, subconsulta1(q)), ejecutarConsulta(b, subconsulta2(q)))) $\wedge_{\text{L}}$
TipoConsulta(q) $\in \{PROJ, INTER, UNION\}$)
}
Post $\equiv$ { res = ejecutarConsulta(b, q) }
Descripción: Dados una consulta c y una base de datos b, devuelve los registros resultantes luego de haber ejecutado la consulta c en la base b, añadiendo las siguientes particularidades dependiendo del Tipo_Consulta(c):

Si TipoConsulta(c) $\in$ {FROM}, devuelve todos los registros contenidos en la tabla correspondiente de b cuyo nombre es nombre_tabla(c).

Si TipoConsulta(c) $\in$ {SELECT}, devuelve todos los registros resultantes al ejecutar la consulta subconsulta1(c) en b, tales que el campo campo1(c) tiene valor valor(c).

Si TipoConsulta(c) $\in$ {MATCH}, devuelve todos los registros resultantes al ejecutar la consulta subconsulta1(c) en b, tales que los campos campo1(c) y campo2(c) tienen el mismo valor.

Si TipoConsulta(c) $\in$ {PROJ}, devuelve todos los registros resultantes al ejecutar la consulta subconsulta1(c) en b, pero que incluyen solamente los campos del conjunto conj_campos(c).

Si TipoConsulta(c) $\in$ {RENAME}, devuelve todos los registros resultantes al ejecutar la consulta subconsulta1(c) en b, tales que el campo campo1(c) será reemplazado por el campo campo2(c).

Si TipoConsulta(c) $\in$ {INTER}, devuelve la intersección entre los registros resultantes de la ejecución de subconsulta1(c) y subconsulta2(c)

Si TipoConsulta(c) $\in$ {UNION}, devuelve la unión entre los registros resultantes de la ejecución de subconsulta1(c) y subconsulta2(c)

Si TipoConsulta(c) $\in$ {PRODUCT}, devuelve el producto cartesiano entre los registros resultantes de la ejecución de subconsulta1(c) y subconsulta2(c).

RegistrosTienenCampo : conj(Registro) $\times$ NombreCampo $\longrightarrow$ bool

RegistrosTienenCampo(c, nc) $\equiv$ ($\forall$ r: Registro)($r \in$ c $\Rightarrow_{\text{L}}$ nc $\in$ campos(r)))

HayCamposEnComun : conj(Registro) $\times$ conj(Registro) $\longrightarrow$ bool

HayCamposEnComun($cr1, cr2$) $\equiv$ ($\exists$ r1: Registro)($\exists$ r2: Registro)($r1 \in cr1 \wedge$ r2 $\in cr2 \wedge$ ($\exists$ nc: nombreCampo)($nc \in campos(r1) \wedge$ (nc $\in campos(r2)$))))

Representación

La base de datos se representa con un diccString, en donde dado un nombre de tabla nt, podemos obtener la tabla correspondiente de la bd asociada con nt.

`BaseDeDatos` **se representa con** `estr`

donde estr es *diccString*(nt:string, t:tabla)

Rep True, porque diccString mantiene consistente toda la informacion interna.

Abs Para todo e: estr, se cumple que: abs(e) = base/dicString::claves(e) = nombreTablas(base) y para todo nt que pertenece a claves(base) implica que diccString::obtener(e, nt) = tabla(base, nt)

Algoritmos

---

iNuevaBd() $\rightarrow$ *res* : *estr*
res $\leftarrow$ diccString::vacio() $//O(1)$

---

agregarTabla(**in/out** *bd*: `estr`, **in** *t*: `tabla`, **in** *nt*: `string`)
diccString::definir(bd, nt, t) $//O(|nt| + copy(t)$

---

borrarTabla(**in/out** bd: `estr`, **in** nt: `string`)

```
if (diccString::definido?(bd, nt) then                    // O(|nt|)
    diccString::borrar(bd, nt)                            // O(|nt|)
end if
```

agregarRegistro(**in/out** bd: `estr`, **in** nt: `string`,**in** r: `registro`)

```
tablaAModificar ← diccString::obtener(bd, nt)             // O(|nt|)
tabla::insertar(tablaAModificar, r)                       // O(copy(r))
```

borrarRegistro(**in/out** bd: `estr`, **in** nt: `string`,**in** v: `string`)

```
tablaAModificar ← diccString::obtener(bd, nt)             // O(|nt|)
tabla::borrar(tablaAModificar, v)                         // O(n), n = #registros)
```

nombreTablas(**in/out** bd: `estr`) → res : $conj(string)$

```
diccString::claves(bd)                                    // O(n * |k|), n = #registros)
```

tabla(**in** bd: `estr`, **in** nt: `string`) → res : $conj(string)$

```
res ← diccString::obtener(bd, nt)                         // O(|nt|)
```

iSelectFrom(**in** bd: `estr`, **in** q: `consulta`) → res : $conj(registro)$

```
conjRes ← vacio()                                                    //O(1)
if c== tabla::clave(tabla) then                                      //O(|c|)
    reg ← *camposXClave(tabla,v)                                     //O(|t| + |v|)
    if !conjLineal::esVacio?(registro::campos(reg))) then            //O(1)
        agregarRapido(conjRes,reg)                                   //O(|v| + |c|)
    end if
else
    nroCampo ← tabla::obtenerNroCampo(tabla,c)                       //O(|c|)
    listaDeItReg ← vacia()                                           //O(1)
    it ← crearItConj(tabla::registros(tabla))                        //O(1)
    while haySiguiente(it) do                                        //O(n)
        if registro::valorEnRegistro(siguiente(it),nroCampo) == v then   //O(|v|)
            agregar(listaDeItReg,it)                                 //O(1)
        end if
        avanzar(itReg)                                               //O(1)
    end while
    itLista ← crearItLista(listaDeItReg)                             //O(1)
    while haySiguiente(itLista) do                                   //O(k) * O(|v| + |c|)
        agregarRapido(conjRes,siguiente(siguiente(itLista)))         //O(|v| + |c|)
        avanzar(itLista)                                             //O(1)
    end while
end if
return conjRes                                                       //O(1)
```

imatchFrom(**in** *tabla*1 : `tabla`, **in** *tabla*2 : `tabla`,**in** *c*1 : `string`,**in** *c*2 : `string`) → *res* : *conj*(*registro*)

```
  campoClave1 ← tabla::clave(t1)                                                    //O(|c|)
  campoClave2 ← tabla::clave(t2)                                                    //O(|c|)
  nroCampo1 ← tabla::obtenerNroCampo(tabla1,camposClave1)                           //O(|c|)
  nroCampo2 ← tabla::obtenerNrocampo(tabla2,campoClave2)                            //O(|c|)
  listaDePunteroReg1 ← vacia()                                                      //O(1)
  listaDePunteroReg2 ← vacia()                                                      //O(1)
  it1 ← crearItConj(tabla::registros(tabla1))                                       //O(1)
  it2 ← crearItConj(tabla::registros(tabla2)                                        //O(1)
  conjRes1 ← vacio()                                                                //O(1)
  conjRes2 ← vacio()                                                                //O(1)
  while haySiguiente(it1) && haySiguiente(it2) do                                   //O(min(n1, n2))
     puntRegMatch1 ← tabla::camposXClave(tabla2, tabla::valorEnRegistro(siguiente(it1),nroCampo1))   O(|v|)
     puntRegMatch2 ← tabla::camposXClave(tabla1, tabla::valorEnRegistro(siguiente(it2),nroCampo2))   O(|v|)
     if !conjLineal::esVacio?(registro::campos(*puntRegMatch1)) then
        agregar(listaDePunteroReg1, ⟨ it1,puntRegMatch1 ∠ )
     end if
     if !conjLineal::esVacio?(registro::campos(*puntRegMatch2)) then
        agregar(listaDePunteroReg1, ⟨ it2,puntRegMatch2 ∠ )
     end if
     avanzar(it1)                                                                   //O(1)
     avanzar(it2)                                                                   //O(1)
  end while
  itLista1 ← crearItLista(listaDeItReg1)                                            //O(1)
  while haySiguiente(itLista1) do                                                   //O(k) * O(|v| + |c|)
     agregarRapido(conjRes1,tabla::mergeReg(siguiente(siguiente(itLista1).it1),*(siguiente(itLista1).puntRegMatch1)))
                                                                                    O(|v| + |c|) + O(1)
     avanzar(itLista1)                                                              O(1)
  end while
  itLista2 ← crearItLista(listaDeItReg2)                                            //O(1)
  while haySiguiente(itLista2) do                                                   //O(k) * O(|v| + |c|)
     agregarRapido(conjRes2,tabla::mergeReg(siguiente(siguiente(itLista2).it2),*(siguiente(itLista2).puntRegMatch2)))
                                                                                    O(|v| + |c|) + O(1)
     avanzar(itLista2)                                                              O(1)
  end while
  if cardinal(conjRes1) > cardinal(conjRes2) then
     return conjRes1
  end if
  return conjRes2
```

iSelectOptimizado(**in** bd: `estr`, **in** q: `consulta`) → res : $conj(registro)$

```
nombreTabla ← consulta::nombreTabla(consulta::subconsultaUno(consulta::subconsulta::subconsultaUno(q)))   //O(|c|)
tablaEnBd ← base_de_datos::tabla(bd,nombreTabla)                                                          //O(|t|)
c1 ← consulta::primerCampo(q))                                                                            //O(1)
c2 ← consulta::primerCampo(consulta::subconsulta1(q))                                                     //O(1)
v1 ← consulta::valor(q)                                                                                   //O(1)
v2 ← consulta::valor(consulta::subconsulta1(q)                                                            //O(1)
nroC2 ← tabla::obtenerNroCampo(tablaEnBd,c2)                                                              //O(|c1|)
conjSelect1 ← base_de_datos::select(tablaEnBd,c1,v1)                                                      //O(|c1| + |v1|)
itConjSelect1 ← itConjLineal::crearIt(conjSelect1)                                                        //O(1)
conjRes = conjLineal::vacio()                                                                             //O(1)
if itConjLineal::haySiguiente(itConjSelect1) then                                                         //O(1)
    reg ← siguiente(itConjSelect1)                                                                        //O(1)
end if
itCamposReg::crearIt(registro::campos(reg))                                                               //O(1)
while itConjLineal::haySiguiente(itCamposReg) do                                                          //O(1)
    if nroC2 == tabla::obtenerNroCampo(tablaEnBd, siguiente(it)) then                                     O(|c2|)
        nroCampoReg ← tabla::obtenerNroCampo(tablaEnBd,siguiente(it)))                                    //O(|c|)
    end if
end while
if nroC2 == nro CampoReg && registro::valorEnRegistro(reg, nroCampoReg) == v2 then                        //O(|v2|)
    conjRes←conjLineal::agregarRapido(reg)                                                                //O(|v| + |c|)
end if
return conjRes                                                                                            //O(1)
```

iSelect(**in** bd: `estr`, **in** q: `consulta`) → res : $conj(registro)$

```
conjRes ← base_de_datos::ejecutarConsulta(consulta::subconsulta1(q))
itConjRes ← conjLineal::crearIt(conjRes)                                                                  //O(1)
while itConjLineal::haySiguiente(itConjRes) do                                                            //O(1)
    huboMatch ← false                                                                                     //O(1)
    itCamposReg = conjLineal::crearIt(registro::campos(itConjLineal::siguiente(itConjRes)))               //O(1)
    reg = itConjLineal::siguiente(itConjRes)
    while itConjLineal::haySiguiente(itCamposReg) do                                                      //O(1)
        nombreCampo ← itConjLineal::siguiente(itCamposReg)
        if nombreCampo == consulta::nombreCampo1(q) reg[nombreCampo] == consulta::valor then //O(|c| + |v|)
            huboMatch←true                                                                                //O(1)
        end if
        itConjLineal::avanzar(itCamposReg)                                                                //O(1)
    end while
    if !huboMatch then                                                                                    //O(1)
        conjLineal::eliminar(conjRes, reg)                                                                //O(k ∗ (|v| + |c|))
    end if
    itConjLineal::avanzar(itConjRes)                                                                      //O(1|)
end while
return conjRes                                                                                            //O(1)
```

iInter(**in** bd: `estr`, **in** q: `consulta`) → res : $conj(registro)$

```
conj1 ← base_de_datos::ejecutarConsulta(consulta::subconsulta1(q))
conj2 ← base_de_datos::ejecutarConsulta(consulta::subconsulta2(q))
conjRes ← base_de_datos::interseccion(conj1,conj2)
return conjRes                                                                                            //O(1)
```

iUnion(**in** bd: `estr`, **in** q: `consulta`) → res : $conj(registro)$

```
conj1 ← base_de_datos::ejecutarConsulta(consulta::subconsulta1(q))
conj2 ← base_de_datos::ejecutarConsulta(consulta::subconsulta2(q))
conjRes ← base_de_datos::union(conj1,conj2)
return conjRes                                                                                            //O(1)
```

iProduct(**in** *bd*: `estr`, **in** *q*: `consulta`) → *res* : *conj*(*registro*)

```
conj1 ← base_de_datos::ejecutarConsulta(consulta::subconsulta1(q))
conj2 ← base_de_datos::ejecutarConsulta(consulta::subconsulta2(q))
conjRes ← base_de_datos::product(conj1,conj2)
return conjRes                                                    //O(1)
```

iDefinirRegistroRenameCampo(**in/out** *reg*: `registro`, **in** *campo1*: `string`, **in** *campo2*: `string`)

```
regRes ← registro::nuevo()
itCamposReg ← itConjLineal::crearIt(registro::campos(reg))
while itConjLineal::haySiguiente(itCamposReg) do
    if itConjLineal::siguiente(itCamposReg) == campo1 then
        registro::definir(regRes, campo2, reg[itConjLineal::siguiente(itCamposReg)])
    end if
    itConjLineal::avanzar(itCamposReg)
end while
return regRes
```

iDefinirRegistroFiltroCampos(**in/out** *reg*: `registro`, **in** *camposFiltro*: `conj(string)`)

```
regRes ← registro::nuevo()
itCamposReg ← itConjLineal::crearIt(registro::campos(reg))
while itConjLineal::haySiguiente(itCamposReg) do
    nombreCampoReg ← itConjLineal::siguiente(itCamposReg)
    if !conjLineal::esVacia?(base_de_datos::filtrarCampo(nombreCampoReg, camposFiltro)) then
        registro::definir(regRes, nombreCampoReg, reg[nombreCampoReg])
    end if
    itConjLineal::avanzar(itCamposReg)
end while
return regRes
```

iInterseccion(**in** *conj*1: `conj(registro)`, **in** *conj*2: `conj(registro)`) → *res* : *conj*(*registro*)

```
conjRes ← conjLineal::vacio()
itConj1 ← itConjLineal::crearIt(conj1)
while itConjLineal::haySiguiente(itConj1) do
    reg ← itConjLineal::siguiente(itConj1)
    if base_de_datos::pertenece(reg, conj2) then
        conjLineal::agregarRapido(conjRes, reg)
    end if
    itConjLineal::avanzar(itConj1)
end while
return conjRes
```

iUnion(**in** *conj*1: `conj(registro)`, **in** *conj*2: `conj(registro)`) → *res* : *conj*(*registro*)

```
conjRes ← conj2
itConj1 ← itConjLineal::crearIt(conj1)
while itConjLineal::haySiguiente(itConj1) do
    reg ← itConjLineal::siguiente(itConj1)
    conjLineal::agregar(conjRes, reg)
    itConjLineal::avanzar(itConj1)
end while
return conjRes
```

iProduct(**in** *conj*1 : `conj(registro)`, **in** *conj*2 : `conj(registro)`) → *res* : *conj*(*registro*)

```
conjRes ← conjLineal::vacio()
itConj1 ← itConjLineal::crearIt(conj1)
while itConjLineal::haySiguiente(itConj1) do
    regsMergeados ← itConjLineal::crearIt(regsMergeados)
    while itConjLineal::haySiguiente(itRegsMergeados) do
        conjLineal::agregar(conjRes, itConjLineal::siguiente(itRegsMergeados))
        itConjLineal::avanzar(itRegsMergeados)
    end while
    itConjLineal::avanzar(itConj1)
end while
return conjRes
```

iMergeRegs(**in** *regToMerge* : `registro`, **in** *conjRegs* : `conj(registro)`) → *res* : *conj*(*registro*)

```
conjRes ← conjLineal::vacio()
itConj ← itConjLineal::crearIt(conjRegs)
while itConjLineal::haySiguiente(itConj) do
    reg ← base_de_datos::mergeDosRegs(regToMerge, itConjLineal::siguiente(itConj))
    conjLineal::agregar(conjRes, reg)
    itConjLineal::avanzar(itConj)
end while
return conjRes
```

iMergeDosRegs(**in** *reg*1 : `registro`, **in** *reg*2 : `registro`) → *res* : *registro*

```
regRes ← reg1
itCampos ← itConjLineal::crearIt(registro::campos(reg2))
while itConjLineal::haySiguiente(itCampos) do
    if conjLineal::esVacia?base_de_datos::interserccion(registro::campos(reg1), registro::campos(reg2)) then
        registro::definir(regRes, itConjLineal::siguiente(itCampos), registro::reg2[itConjLineal::siguiente(itCampos)])
    end if
    itConjLineal::avanzar(itCampos)
end while
return regRes
```

iInterseccion(**in** *conjCampos*1 : `conj(string)`, **in** *conjCampos*2 : `conj(string)`) → *res* : *conj*(*string*)

```
conjRes ← conjLineal::vacio()
itCampos1 ← itConjLineal::crearIt(conjCampos1)
while itConjLineal::haySiguiente(itCampos1) do
    campo ← itConjLineal::siguiente(itCampos1)
    if base_de_datos::pertenece(campo, conjCampos2) then
        conjLineal::agregarRapido(conjRes, campo)
    end if
    itConjLineal::avanzar(itCampos1)
end while
return conjRes
```

iFiltrarCampo(**in** *campo*1 : `string`, **in** *conjCampos*2 : `conj(string)`) → *res* : *string*

```
conjRes ← conjLineal::vacio()
if base_de_datos::pertenece(campo1, conjCampos2) then
    conjLineal::agregarRapido(conjRes, campo1)
end if
return conjRes
```

iPertenece(**in** *reg* : `registro`, **in** *conj* : `conj(registro)`) → *res* : *bool*

```
itConj ← itConjLineal::crearIt(conj)
while itConjLineal::haySiguiente(itConj) && reg ≠ itConjLineal::siguiente(itConj) do
    itConjLineal::avanzar(itConj)
end while
return itConjLineal::haySiguiente(itConj)
```

iPertenece(**in** *nc* : `string`, **in** *conj* : `conj(string)`) → *res* : *bool*

```
itConj ← itConjLineal::crearIt(conj)
while itConjLineal::haySiguiente(itConj) && nc ≠ itConjLineal::siguiente(itConj) do
    itConjLineal::avanzar(itConj)
end while
return itConjLineal::haySiguiente(itConj)
```

iSelectProductOptimizado(**in** *bd* : `estr`, **in** *q* : `consulta`) → *res* : *conj*(*registro*)

```
tabla1 ← base_de_datos::tabla(bd, consulta::nombreTabla(consulta::subconsultaUno(consulta::subconsultaUno(q)))) //O(|t|)
tabla2 ← base_de_datos::tabla(bd, consulta::nombreTabla(consulta::subconsultaDos(consulta::subconsultaUno(q)))) //O(|t|)
conjRes ← conjLineal::vacio()                                                          //O(1)
regTabla1 = *tabla::camposXClave(tabla1, consulta::valor(q))                           //O(|v|)
itConj = conjLineal::crearIt(tabla::registros(tabla2))                                 //O(n2 * (|v| + |c|)
while (itConjLineal::haySiguiente(itConj)) do                                          //O(|n2|)
    regRes ← mergeReg(regTabla1, itConjLineal::siguiente(itConj))                      //O(|v| + |c|)
    conjLineal::agregarRapido(conjRes, regRes)                                         //O(|v| + |c|)
end while
return conjRes                                                                         //O(1)
```

iMatch(**in** *bd* : `estr`, **in** *q* : `consulta`) → *res* : *conj*(*registro*)

```
conjRes ← base_de_datos::ejecutarConsulta(consulta::subconsulta1(q))    //O(|ejecutarConsulta(subconsulta1(q))|)
itConjRes ← conjLineal::crearIt(conjRes)                                //O(|1|)
while (itConjLineal::haySiguiente(itConjRes)) do                        //O(|1|)
    reg ← itConjLineal::siguiente(itConjRes)                            //O(|1|)
    if (reg[consulta::primerCampo(q)] != reg[consulta::segundoCampo(q]) then
        conjLineal::eliminar(conjRes, reg)                              //O(k * (|c| + |v|))
    end if
    itConjLineal::avanzar(itConjRes)                                    //O(|1|)
end while
return conjRes                                                          //O(|1|)
```

iProj(**in** *bd* : `estr`, **in** *q* : `consulta`) → *res* : *conj*(*registro*)

```
conjRes ← base_de_datos::ejecutarConsulta(consulta::subconsulta1(q))    //O(|ejecutarConsulta(subconsulta1(q))|)
itConjRes ← conjLineal::crearIt(conjRes)                                //O(|1|)
camposDeConsulta ← consulta::conjCampos(q)                              // O(|1|)
while (itConjLineal::haySiguiente(itConjRes)) do                        //O(|1|)
    regRes ← base_de_datos::definirRegistroFiltroCampos(itConjLineal::siguiente(itConjRes), camposDeConsulta)
    conjLineal::agregarRapido(conjRes, regRes)                          //O(|c| + |v|)
    itConjLineal::avanzar(itConjRes)                                    //O(|1|)
end while
return conjRes                                                          //O(|1|)
```

iRename(**in** bd: `estr`, **in** q: `consulta`) $\rightarrow$ res : $conj(registro)$

conjRes $\leftarrow$ base_de_datos::ejecutarConsulta(consulta::subconsulta1(q)) $//O(|ejecutarConsulta(subconsulta1(q))|)$
itConjRes $\leftarrow$ conjLineal::crearIt(conjRes) $//O(|1|)$
camposDeConsulta $\leftarrow$ consulta::conjCampos(q) $// O(|1|)$
while (itConjLineal::haySiguiente(itConjRes)) **do** $// O(|1|)$
    regRes $\leftarrow$ base_de_datos::definirRegistroRenameCampo(itConjLineal::siguiente(itConjRes), consulta::nombreCampo1(q), consulta::nombreCampo2(q)) $// O(|1|)$
    conjLineal::agregarRapido(conjRes, regRes) $//O(|c| + |v|)$
    itConjLineal::avanzar(itConjRes) $//O(|1|)$
end while
return conjRes $//O(|1|)$

iEjecutarConsulta(**in** bd : `estr`, **in** q : `consulta`) → res : $conj(registro)$

```
conjRes ← conjLineal::vacio()                                                    //O(|1|)
switch(consulta::tipoConsulta(q))                                                //O(|1|)
{
CASE 'FROM':                                                                     //O(|1|)
tablaEnBd ← base_de_datos::tabla(bd, consulta::nombreTabla(q))                   //O|t|
conjRes ← tabla::registros(tablaEnBd)
break;                                                                           //O(|1|)
CASE 'SELECT':                                                                   //O(|1|)
if (consulta::tipoConsulta(consulta::subconsultaUno(q)) == 'FROM') then          // O(|1|)
    tablaEnBd ← base_de_datos::tabla(bd, consulta::nombreTabla(consulta::subconsultaUno(q)))   //O|t|
    conjRes ← selectFrom(tablaEnBd, consulta::primerCampo(q), consulta::valor(q))             //O|?|
else if (consulta::tipoConsulta(consulta::subconsultaUno(q)) == 'SELECT' && consulta::tipoConsulta(consulta::subconsultaUno(consulta::subconsultaUno(q))) == 'FROM') then   //O|1|
    tablaEnBd ← base_de_datos::tabla(bd, consulta::nombreTabla(consulta::subconsultaUno(consulta::subconsultaUno(q))))
                                                                                 //O|t|
    if (base_de_datos::esClaveTabla(tablaEnBd, consulta::primerCampo(consulta::subconsultaUno(q))) && !base_de_datos::esClaveTabla(tablaEnBd, consulta::primerCampo(consulta::subconsultaUno(consulta::subconsultaUno(q)))))
    then                                                                         //O|c|
        conjRes ← base_de_datos::selectOptimizado(bd, q)
    end if
else if (consulta::tipoConsulta(consulta::subconsultaUno(q)) == 'PRODUCT' && consulta::tipoConsulta(consulta::subconsultaUno(consulta::subconsultaUno(q))) == 'FROM' && consulta::tipoConsulta(consulta::subconsultaDos(consulta::subconsultaUno(q))) == 'FROM') then   //O|1|
    conjRes ← selectProductOptimizado(bd, q)
else
    conjRes ← select(bd, q)
end if
break;                                                                           //O(|1|)
CASE 'MATCH':                                                                    //O(|1|)
if (consulta::tipoConsulta(consulta::subconsulta1(q)) == 'PRODUCT' && consulta::tipoConsulta(consulta::subconsultaUno(consulta::subconsultaUno(q))) == 'FROM' && consulta::tipoConsulta(consulta::subconsultaDos(consulta::subconsultaUno(q))) == 'FROM') then
    tablaEnBd1 ← base_de_datos::tabla(bd, consulta::nombreTabla(consulta::subconsultaUno(consulta::subconsultaUno(q))))

    tablaEnBd2 ← base_de_datos::tabla(bd, consulta::nombreTabla(consulta::subconsultaDos(consulta::subconsultaUno(q))))

    conjRes ← matchFrom(tablaEnBd1, tablaEnBd2, consulta::nombreCampo1(consulta::subconsulta1(q)), consulta::nombreCampo2(consulta::subconsulta1(q)))
else
    conjRes ← base_de_datos::match(bd, q)
end if
break;                                                                           //O(|1|)
CASE 'PROJ':                                                                     //O(|1|)
conjRes ← base_de_datos::proj(bd, q)
break;                                                                           //O(|1|)
CASE 'RENAME':                                                                   //O(|1|)
conjRes ← base_de_datos::rename(bd, q)
break;                                                                           //O(|1|)
CASE 'INTER':                                                                    //O(|1|)
conjRes ← base_de_datos::inter(bd, q)
break;                                                                           //O(|1|)
CASE 'UNION':                                                                    //O(|1|)
conjRes ← base_de_datos::union(bd, q)
break;                                                                           //O(|1|)
CASE 'PRODUCT':                                                                  //O(|1|)
conjRes ← base_de_datos::product(bd, q)
CASE 'DEFAULT':                                                                  //O(|1|)
conjRes ← conjLineal::vacio()
break;                                                                           //O(|1|)
}
return conjRes;                                                                  //O(|1|)
```

2. Modulo Consulta

El modulo Consulta provee operaciones básicas para generar una consulta. Permite construir una consulta a partir de una tabla.

Para describir la complejidad de las operaciones, el costo de copiar un $s \in string$ e igualar $s_1, s_2 \in string$ es igual a $O(|s|)$). Y el costo de copiar una consulta q sera expresada como $copy(q)$ y sera tomado como la copia recursiva de todos los campos correspondiente en cada una de las instancias de la recursion.

Interfaz

se explica con: Consulta

usa: Registro, Tabla, Bool, Nat, String

género: `consulta`

FROM(**in** nt : $nombreTabla$) $\rightarrow$ res : $Consulta$
Pre $\equiv$ {true}
Post $\equiv$ {res = FROM(nt)}
Complejidad: $O(|nt|)$
Descripción: Crea una consulta c a partir de un nombreTabla nt, que tiene las siguientes particularidades: TipoConsulta(c): FROM, NombreTabla(c): nt

SELECT(**in** c : $Consulta$, **in** nc : $nombreCampo$, **in** v : $Valor$) $\rightarrow$ res : $Consulta$
Pre $\equiv$ {true}
Post $\equiv$ {res = SELECT(c, nc, v)}
Complejidad: $O(|nc| + |v| + copy(c))$
Descripción: Dada una consulta c, un nombreCampo nc y un valor v, SELECT es una nueva consulta que contiene las siguientes particularidades: TipoConsulta(res) : SELECT, PrimerCampo(res): nombre, Valor(res): v, SubconsultaUno(res): c

MATCH(**in** c : $Consulta$, **in** $nc1$: $nombreCampo$, **in** $nc2$: $nombreCampo$) $\rightarrow$ res : $Consulta$
Pre $\equiv$ {true}
Post $\equiv$ {res = MATCH(c0, nc1, nc2)}
Complejidad: $O(|nc1| + |nc2| + copy(c))$
Descripción: Dada una consulta c y dos nombreCampo nc1 y nc2 , MATCH es una nueva consulta que contiene las siguientes particularidades: TipoConsulta(res) : MATCH, PrimerCampo(res): nc1, SegundoCampo(res): nc2, SubconsultaUno(res): c

PROJ(**in** c : $Consulta$, **in** cnc : $conj(nombreCampo)$) $\rightarrow$ res : $Consulta$
Pre $\equiv$ {true}
Post $\equiv$ {c = PROJ(c, cnc)}
Complejidad: $O(|cnc| + copy(c))$
Descripción: Dada una consulta c y un conj(nombreCampo) campos, PROJ es una nueva consulta que contiene las siguientes particularidades: TipoConsulta(nc) : PROJ, Conj_Campos(nc): campos, SubconsultaUno(nc): c

RENAME(**in** c : $Consulta$, **in** $nc1$: $nombreCampo$, **in** $nc2$: $nombreCampo$) $\rightarrow$ res : $Consulta$
Pre $\equiv$ {true}
Post $\equiv$ {c = RENAME(c, nc1, nc2)}
Complejidad: $O(|nc1| + |nc2| + copy(c))$
Descripción: Dada una consulta c y dos nombresCampo nc1 y nc2, RENAME es una nueva consulta que contiene las siguientes particularidades: TipoConsulta(res) : RENAME, PrimerCampo(res) : nc1, SegundoCampo(res) : nc2, SubconsultaUno(res): c

INTER(**in** $q1$: $consulta$, **in** $q2$: $Consulta$) $\rightarrow$ res : $Consulta$
Pre $\equiv$ {true}
Post $\equiv$ {c = INTER(q1, q2)}
Complejidad: $O(copy(q1) + (copy(q1))$
Descripción: Dada una consulta c y dos nombresCampo nc1 y nc2, INTER es una nueva consulta que contiene las siguientes particularidades: TipoConsulta(res) : INTER, SubconsultaUno(res): q1 SubconsultaDos(res): q2

UNION(**in** $q1$: $consulta$, **in** $q2$: $Consulta$) $\rightarrow$ res : $Consulta$
Pre $\equiv$ {true}
Post $\equiv$ {c = UNION(q1, q2)}
Complejidad: $O(copy(q1) + (copy(q1))$
Descripción: Dada una consulta c y dos nombresCampo nc1 y nc2, UNION es una nueva consulta que contiene las siguientes particularidades: TipoConsulta(res) : UNION, SubconsultaUno(res): q1 SubconsultaDos(res): q2

PRODUCT(**in** $q1$: *consulta*, **in** $q2$: *Consulta*) $\rightarrow$ *res* : *Consulta*
Pre $\equiv$ {true}
Post $\equiv$ {c = PRODUCT(q1, q2)}
Complejidad: $O(copy(q1) + (copy(q1))$
Descripción: Dada una consulta c y dos nombresCampo nc1 y nc2, UNION es una nueva consulta que contiene las siguientes particularidades: TipoConsulta(res) : PRODUCT, SubconsultaUno(res): q1 SubconsultaDos(res): q2

TipoConsulta(**in** c: *Consulta*) $\rightarrow$ *res* : `tipoConsulta`
Pre $\equiv$ {true}
Post $\equiv$ {res = tipo_consulta(c)}
Complejidad: $O(1)$
Descripción: Dada una consulta, devuelve su tipo

NombreTabla(**in** c: *Consulta*) $\rightarrow$ *res* : `nombreTabla`
Pre $\equiv$ {tipo_consulta(c) $\in$ {FROM}}
Post $\equiv$ {res = nombre_tabla(c)}
Complejidad: $O(copy(nombreTabla))$
Descripción: Dada una consulta de tipo FROM, devuelve la tabla asociada a dicha consulta

PrimerCampo(**in** c: *Consulta*) $\rightarrow$ *res* : `nombre_campo`
Pre $\equiv$ {tipo_consulta(c) $\in$ {SELECT, MATCH, RENAME}}
Post $\equiv$ {res = campo1(c)}
Complejidad: $O(copy(nombreCampo))$
Descripción: Dada una consulta de tipo FROM, MATCH O RENAME, devuelve el primer parámetro de tipo nombreCampo asociado a dicha consulta

SegundoCampo(**in** c: *Consulta*) $\rightarrow$ *res* : `nombre_campo`
Pre $\equiv$ {tipo_consulta(c) $\in$ {MATCH, RENAME}}
Post $\equiv$ {res = campo2(c)}
Complejidad: $O(copy(nombreCampo))$
Descripción: Dada una consulta de tipo MATCH O RENAME, devuelve el segundo parámetro de tipo nombreCampo asociado a dicha consulta

Valor(**in** c: *Consulta*) $\rightarrow$ *res* : `valor`
Pre $\equiv$ {tipo_consulta(c) $\in$ {SELECT}}
Post $\equiv$ {res = valor(c)}
Complejidad: $O(copy(valor))$
Descripción: Dada una consulta de tipo SELECT(consulta, nombreCampo, valor), devuelve el valor asociado a dicha consulta

ConjCampos(**in** c: *Consulta*) $\rightarrow$ *res* : `Conj(nombreCampo)`
Pre $\equiv$ {tipo_consulta(c) $\in$ {PROJ}}
Post $\equiv$ {res = conj_campos(c)}
Complejidad: $O(copy(conj(nombreCampo)))$
Descripción: Dada una consulta de tipo PROJ(consulta, conjCampos), devuelve el conjunto de campos asociado a dicha consulta

SubconsultaUno(**in/out** c: *Consulta*)
Pre $\equiv$ {c = c0 $\wedge$ tipo_consulta(c0) $\in$ {SELECT, MATCH, PROJ, RENAME, INTER, UNION, PRODUCT}}
Post $\equiv$ {c = subconsulta1(c0)}
Complejidad: $O(copy(consulta))$
Descripción: Dada una consulta de tipo SELECT, MATCH, PROJ, RENAME, INTER, UNION O PRODUCT, devuelve la primer subconsulta asociada a dicha consulta

SubconsultaDos(**in/out** c: *Consulta*)
Pre $\equiv$ {c = c0 $\wedge$ tipo_consulta(c0) $\in$ {INTER, UNION, PRODUCT}}
Post $\equiv$ {c = subconsulta2(c0)}
Complejidad: $O(copy(consulta))$
Descripción: Dada una consulta de tipo INTER, UNION, PRODUCT, nos da la segunda subconsulta asociada a dicha consulta.

Representación

La consulta se representan de forma recursiva, con los campos correspondientes a la consulta adecuada.

`consulta` se representa con `estr`

donde `estr` es `tupla`(*tipoConsulta*: `string`,
nombreTabla: `string`,
campo1: `string`,
campo2: `string`,
valor: `string`,
conjCampos: `conj(string)`,
subConsulta1: `consulta`,
subConsulta2: `consulta`)

Rep

- Para todo c de tipo consulta, el campo c.tipoConsulta debe pertencer al conjunto {FROM, SELECT, MATCH, RENAME, PROJ, INTER, UNION, PRODUCT}
- Si c.tipoConsulta es FROM: tendra que tener un c.nombreTabla valido, y todos los demas campos seran nulos (segun su tipo)
- Si c.tipoConsulta es SELECT: tendra que tener un c.nombreCampo1 y c.valor validos, y todos los demas campos seran nulos (segun su tipo)
- Si c.tipoConsulta es MATCH: tendra que tener un c.nombreCampo1 y c.nombreCampo2 validos, y todos los demas campos seran nulos (segun su tipo)
- Si c.tipoConsulta es PROJ: tendra que tener un c.conjCampos valido, y todos los demas campos seran nulos (segun su tipo)
- Si c.tipoConsulta es RENAME: tendra que tener un c.nombreCampo1 y c.nombreCampo2 validos, y todos los demas campos seran nulos (segun su tipo)
- Si c.tipoConsulta es INTER: tendra que tener un c.subconsulta1 y c.subconsulta2 validos, y todos los demas campos seran nulos (segun su tipo)
- Si c.tipoConsulta es UNION: tendra que tener un c.subconsulta1 y c.subconsulta2 validos, y todos los demas campos seran nulos (segun su tipo)
- Si c.tipoConsulta es PRODUCT: tendra que tener un c.subconsulta1 y c.subconsulta2 validos, y todos los demas campos seran nulos (segun su tipo)

Abs

Abs : estr e $\longrightarrow$ consulta {Rep(e)}

Abs(e) $\equiv$ c: consulta | tipo_consulta(c) $=_{\text{obs}}$ e.tipoConsulta
nombre_tabla(c) $=_{\text{obs}}$ e.nombreTabla
campo$_1$(c)$=_{\text{obs}}$ e.*campo*1
campo$_2$(c)$=_{\text{obs}}$ e.*campo*2
valor(c)$=_{\text{obs}}$ e.*valor*
conj_campos(c)$=_{\text{obs}}$ e.*conjCampos*
subconsulta$_1$(c)$=_{\text{obs}}$ e.*subconsulta*1
subconsulta$_2$(c)$=_{\text{obs}}$ e.*subconsulta*2

Algoritmos

---

iFrom(**in** nt : `nombreTabla`) $\rightarrow$ res : estr

1: $res \leftarrow$ ⟨ FROM, nt, string::Vacio(), string::Vacio(), string::Vacio(), conj::Vacio(), NULL, NULL ⟩

Complejidad: $O(|nt|)$

---

iSelect(**in** c : `estr`, **in** nc : `nombreCampo`, **in** v : `valor`,) $\rightarrow$ res : estr

1: $res \leftarrow$ ⟨ SELECT, string::Vacio(), nc, string::Vacio(), v, conj::Vacio(), c, NULL ⟩

Complejidad: $O(|nc| + |v| + copy(c))$

---

iMatch(**in** c : `estr`, **in** $nc1$: `nombreCampo`, **in** $nc2$: `nombreCampo`) $\rightarrow$ res : estr

1: $res \leftarrow$ ⟨ MATCH, string::Vacio(), nc1, nc2, string::Vacio(), conj::Vacio(), c, NULL ⟩

Complejidad: $O(|nc1| + |nc2| + copy(c))$

---

iProj(**in** c : `estr`, **in** cnc : `conj(nombreCampo)`) $\rightarrow$ res : estr

1: $res \leftarrow$ ⟨ PROJ, string::Vacio(), string::Vacio(), string::Vacio(), string::Vacio(), cnc, c, NULL ⟩

Complejidad: $O(|cnc| + copy(c))$

---

iRename(**in** c : `estr`, **in** $nc1$: `nombreCampo`, **in** $nc2$: `nombreCampo`) $\rightarrow$ res : estr

1: $res \leftarrow$ ⟨ RENAME, string::Vacio(), nc1, nc2, string::Vacio(), conj::Vacio(), c, NULL ⟩

Complejidad: $O(|nc1| + |nc2| + copy(c))$

---

iInter(**in** $c1$: `estr`, **in** $c2$: `estr`) $\rightarrow$ res : estr

1: $res \leftarrow$ ⟨ INTER, string::Vacio(), string::Vacio(), string::Vacio(), string::Vacio(), conj::Vacio(), c1, c2 ⟩

Complejidad: $O(copy(c1) + copy(c2))$

---

iUnion(**in** $c1$: `estr`, **in** $c2$: `estr`) $\rightarrow$ res : estr

1: $res \leftarrow$ ⟨ UNION, string::Vacio(), string::Vacio(), string::Vacio(), string::Vacio(), conj::Vacio(), c1, c2 ⟩

Complejidad: $O(copy(c1) + copy(c2))$

---

iProduct(**in** $c1$: `estr`, **in** $c2$: `estr`) $\rightarrow$ res : estr

1: $res \leftarrow$ ⟨ PRODUCT, string::Vacio(), string::Vacio(), string::Vacio(), string::Vacio(), conj::Vacio(), c1, c2 ⟩

Complejidad: $O(copy(c1) + copy(c2))$

---

iTipoConsulta(**in** c : `estr`) $\rightarrow$ res : tipoConsulta

1: $res \leftarrow$ c.tipoConsulta

Complejidad: $O(tipoConsulta) = O(1)$

---

iNombreTabla(**in** c : `estr`) $\rightarrow$ res : nombreTabla

1: $res \leftarrow$ c.nombreTabla

Complejidad: $O(copy(nombreTabla))$

---

iPrimerCampo(**in** c : `estr`) $\rightarrow$ res : nombreCampo

1: $res \leftarrow$ c.campo1

Complejidad: $O(copy(nombreCampo))$

---

iSegundoCampo(**in** c : `estr`) $\rightarrow$ res : nombreCampo

1: $res \leftarrow$ c.campo2

Complejidad: $O(copy(nombreCampo))$

---

iValor(**in** c: `estr`) $\rightarrow$ res : valor

1: $res \leftarrow$ c.valor

Complejidad: $O(copy(valor))$

iConjCampos(**in** c: `estr`) $\rightarrow$ res : conj(nombreCampo)

1: $res \leftarrow$ c.conjCampos

Complejidad: $O(copy(conj(nombreCampo)))$

iSubconsulta1(**in** c: `estr`) $\rightarrow$ res : consulta

1: $res \leftarrow$ c.subconsulta1

Complejidad: $O(copy(consulta))$

iSubconsulta2(**in** c: `estr`) $\rightarrow$ res : consulta

1: $res \leftarrow$ c.subconsulta2

Complejidad: $O(copy(consulta))$

copiarConsulta(**in** c: `estr`) $\rightarrow$ res : consulta

1: $res.nombreTabla \leftarrow$ c.nombreTabla
2: $res.campo1 \leftarrow$ c.campo1
3: $res.campo2 \leftarrow$ c.campo2
4: $res.valor \leftarrow$ c.valor
5: $res.conjCampos \leftarrow$ c.conjCampos
6: $res.subConsulta1 \leftarrow$ c.subConsulta1
7: $res.subConsulta2 \leftarrow$ c.subConsulta2

Complejidad: $copy(c) = O(|c.nombreTabla|, |c.campo1|, |c.campo2|, |c.valor|, copy(c.conjCampos), copy(c.subConsulta1), copy(c.subConsulta2))$

3. Modulo Registro

El modulo Registro provee operaciones básicas a realizarse sobre un registro. Permite crear un nuevo registro, obtener los campos de un registro, y obtener el valor de uno de los campos de un registro. Para describir la complejidad de las operaciones, el costo de copiar un $s \in string$ e igualar $s_1, s_2 \in string$ es igual a $O(|s|)$).

Interfaz

se explica con: REGISTRO

usa: REGISTRO, TABLA, CONSULTA, BOOL, NAT, STRING

género: `registro`

NUEVO() $\rightarrow res$: *Registro*
Pre $\equiv$ {true}
Post $\equiv$ {res = Registro}
Complejidad: $O(1)$
Descripción: Crea un nueva registro vacio.
Aliasing: No Aplica

DEFINIR(**in/out** r: *Registro*, **in** nc: *nombreCampo*, **in** v: $Valor$)
Pre $\equiv$ {r = r0 $\wedge$ (nc $\notin$ campos(r))}
Post $\equiv$ {res = definir(r0,nc,v)}
Complejidad: $O(|nc| + |v|)$
Descripción: Define un nuevo campo y su valor correspondiente dentro de un registro.
Aliasing: r es modificable

CAMPOS(**in** r: *Registro*) $\rightarrow res$: $Conj(nombreCampo)$
Pre $\equiv$ {true}
Post $\equiv$ {res = campos(r)}
Complejidad: $O(|c|)$
Descripción: Devuelve las campos de un registro (conj(nombreCampo)).
Aliasing: no aplica

• [•](**in** r: *Registro*, **in** nc: $NombreCampo$) $\rightarrow res$: $Valor$
Pre $\equiv$ {nc $\in$ campos(r)}
Post $\equiv$ {res = r[nc]}
Complejidad: $O(|c| + |v|)$
Descripción: Devuelve el valor del campo de un registro.
Aliasing: no aplica

VALORXNROCAMPO(**in** r: *Registro*, **in** $nroCampo$: nat) $\rightarrow res$: $Valor$
Pre $\equiv$ {nroCampo $\leq$ #campos(r)}
Post $\equiv$ {res = r[ordenLexicografico(r, nc)]}
Complejidad: $O(|v|)$
Descripción: Devuelve el valor del campo de un registro.
Aliasing: no aplica

ordenLexicografico : registro r $\times$ nombreCampo nc $\longrightarrow$ nat $\quad$ {nc $\in$ campos(r)}

ordenLexicografico(r, nc) $\equiv$ auxOrdenLex(campos(r), nc, 1)

auxOrdenLex : conj(nombreCampo) cnc $\times$ nombreCampo nc $\times$ nat $\longrightarrow$ nat $\quad$ {nc $\in$ cnc}

auxOrdenLex(cnc, nc, n) $\equiv$ **if** ($\emptyset$?(cnc)) **then**
n
else
if (dameUno(cnc) $<$ nc) **then**
auxOrdenLex(sinUno(cnc), nc, $n + 1$)
else
auxOrdenLex(sinUno(cnc), nc, n)
fi
fi

• $<$ • : string $\times$ string $\longrightarrow$ bool

• $<$ • ($s1, s2$) $\equiv$ **if** (vacia?($s1$) $\vee_{\text{L}}$ vacia?($s2$)) **then** vacia?(s1) **else** prim($s1$) $<$ prim($s2$) (fin($s2$) $<$ fin($s2$) **fi**

Representación

El registro se representa con un conjunto de tuplas que guardan toda la informacion requerida. Para describir la complejidad de las operaciones, el costo de copiar un $s \in string$ e igualar $s_1, s_2 \in string$ es igual a $O(|s|)$).

`registro` se representa con `estr`

donde estr es $conj(Tupla <$ idCampo: nat, nc: $string$, valor: $string >$)

Rep

- No existe una tupla que tenga el mismo nc e idCampo que otra dentro del conjunto
- Cada idCampo de cualquier tupla tiene que estar mapeado correctamente con el orden lexicografico asignado a cada nc, o sea que para todo tupla, tiene que cumplirse que el idCampo sea el correspondiente a nc

Abs

Abs : estr e $\longrightarrow$ registro $\{\text{Rep}(e)\}$

Abs(e) $\equiv$ r: registro | campos(r) $=_{\text{obs}}$ filtroDeCampos(e,$\emptyset$)
$(\forall nc : nombre\_campo)(nc \in campos(r) \Rightarrow r[nc] =_{\text{obs}} buscarNC(e, nc)$

filtroDeCampos : estr $\times$ conj(string) $\longrightarrow$ conj(string)
filtroDeCampos(e, res) $\equiv$ **if** (vacia?(e)) **then** res **else** filtroDeCampos(sinUno(e), Ag(res, dameUno(e).nc)) **fi**

buscarNC : estr $\times$ nombre_campo $\longrightarrow$ valor $\{(\exists\ tupla :< nat, string, string >)(tupla \in estr \Rightarrow nc = tupla.nc\}$
buscarNC(e, nc) $\equiv$ **if** (dameUno(e).nc = nc) **then** dameUno(e).valor **else** buscarNC(sinUno(e), nc) **fi**

Algoritmos

---

iNuevo() $\rightarrow$ res : estr

```
res ← conj::vacio()                                                    //O(1)
```

---

iDefinir(**in/out** r : `estr`, **in** nc : `nombreCampo`,**in** v : `valor`)

```
nroCampo ← 0                                                           //O(1)
itCampos ← conj::crearIt(r)                                            //O(1)
while conj::haySiguiente(itCampos) do                                  //O(|nc|)
    if menorLexicografico(siguiente(itCampos), nombreCampo then        //O(|nc|)
        nroCampo ← nroCampo + 1                                        //O(1)
    end if
    conj::avanzar(itCampos)                                            //O(1)
end while
conj::agregar(r, ⟨ nroCampo, nc, v ⟩)                                  //O(|nc| + |v|)
```

---

iMenorLexicografico(**in** $s1$: `string`, **in** $s2$: `string` $\rightarrow$ res : bool

```
menor ← false                                                          //O(1)
it1 ← string::itCrear(s1)                                              //O(1)
it2 ← string::itCrear(s2)                                              //O(1)
while string::haySiguiente(it1) && string::haySiguiente(it2) do        //O(min(|s1|, |s2|)
    if siguiente(it1) < siguiente(it2) then                            //O(1)
        menor ← true                                                   //O(1)
    end if
    string::avanzar(it1)                                               //O(1)
    string::avanzar(it2)                                               //O(1)
end while
return menor                                                           //O(1)
```

iCampos(**in** r : `estr`) → res : conj(nombreCampo)

```
itCampos ← conj::crearIt(r))                                          //O(1)
while conj::haySiguiente(itCampos) do                                 //O(|c|)
    res ← conj::agregar(r, siguiente(itCampos))                       //O(|c|)
    avanzar(itCampos)                                                 //O(1)
end while
return res                                                            //O(1)
```

i•[•](**in** r : `estr`, **in** nc : `nombreCampo` → v :valor

```
itCampos ← conj::crearIt(r))                                          //O(1)
while conj::haySiguiente(itCampos) do                                 //O(|v| + |c|)
    if nc == siguiente(itCampos).nc then                              //O(|v|)
        res ← siguiente(itCampos).valor                               //O(|v|)
    end if
    avanzar(itCampos)                                                 //O(1)
end while
return res                                                            //O(1)
```

iValorXNroCampo(**in** r : `estr`, **in** $nroCampo$: `nat` → res : valor)

```
itCampos ← crearItConj(r)                                             //O(1)
while (haySiguiente(itCampos)) do                                     //O(1)
    if (siguiente(itCampos).idCampo == nroCampo) then                 //O(1)
        valor ← siguiente(itCampos).valor                             //O(|v|)
    end if
    avanzar(itCampos)                                                 //O(1)
end while
return valor                                                          //O(1)
```

4. Modulo Tabla

El modulo Tabla provee operaciones básicas a realizarse sobre una tabla. Permite crear una tabla, agregar o eliminar un registro de una tabla, y brinda operaciones para obtener la clave de una tabla, obtener las columnas(conj(nombreCampo)) y las filas(conj(Registro)) de una tabla. Para describir la complejidad de las operaciones, el costo de copiar un $s \in string$ e igualar $s_1, s_2 \in string$ es igual a $O(|s|)$).

Interfaz

se explica con: TABLA

usa: REGISTRO, CONSULTA, BOOL, NAT, CONJUNTOLINEAL

género: `tabla`.

NUEVATABLA(**in** c: $conj(nombreTabla)$, **in** nt: $nombreTabla$) $\rightarrow$ res : `Tabla`
Pre $\equiv$ {nt $\in$ c}
Post $\equiv$ {res = nueva(t, nt)}
Complejidad: $O(1)$
Descripción: Crea una tabla a una base de datos.
Aliasing: No aplica

INSERTAR(**in/out** t: $Tabla$, **in** r: $Registro$)
Pre $\equiv$ {t = t0 $\wedge$ campos(t) = campos(r)}
Post $\equiv$ {t = insertar(t0, r)}
Complejidad: $O()$
Descripción: Dados una tabla y un registro, inserta el registro en dicha tabla. En caso de que el registro ya exista en la tabla, lo sobreescribe

BORRAR(**in/out** t: $Tabla$, **in** v: $Valor$)
Pre $\equiv$ {t = t0}
Post $\equiv$ {($\neg$ValorEnTabla(t, v) $\wedge$ t=t0) $\vee$ (ValorEnTabla(t, v) $\wedge$ t=borrar(t, v))}
Complejidad: $O()$
Descripción: Dados una tabla y un valor, se fija si dicho valor coincide con alguno de los valores del campo clave de la tabla.En caso afirmativo borra el registro que contenga dicho valor, de lo contrario devuelve la tabla sin modificaciones.

CAMPOS(**in** t: $Tabla$) $\rightarrow$ res : `Conj(nombreCampo)`
Pre $\equiv$ {true}
Post $\equiv$ {res = campos(t)}
Complejidad: $O()$
Descripción: Devuelve las columnas(conj(nombreCampo)) de la tabla.

REGISTROS(**in** t: $Tabla$) $\rightarrow$ res : `Conj(Registro)`
Pre $\equiv$ {true}
Post $\equiv$ {res = registros(t)}
Complejidad: $O()$
Descripción: Devuelve las filas(conj(Registro)) de la tabla.

CLAVE(**in** t: $Tabla$) $\rightarrow$ res : `nombreCampo`
Pre $\equiv$ {true}
Post $\equiv$ {res = clave(t)}
Complejidad: $O()$
Descripción: Devuelve el nombre del campo declarado como clave de la tabla.

OBTENERNROCAMPO(**in** t: $Tabla$ **in** nc: `string`) $\rightarrow$ res : `nat`
Pre $\equiv$ {(Existe r: registro) (r pertenece a t.registros()) yLuego nc pertenece a r)}
Post $\equiv$ {(ParaTodo r: registro) (r pertenece a t.registros() implicaLuego res = ordenLexicografico(r, nc)) }
Complejidad: $O(|nc|)$
Descripción: Dados una tabla t y un nombreCampo c, devuelve el nat asociado al mapeo existente según un orden lexicografico que poseen los registros.

CAMPOSXCLAVE(**in** t: $Tabla$ **in** $valor$: `string`) $\rightarrow$ res : `puntero(registro)`
Pre $\equiv$ {true}
Post $\equiv$ {(Existe r:registro) (r pertenece a t.registros() y (Existe nc:nombreCampo) (nombreCampo pertenece a r.campos yLuego r[nc] = v implicaLuego puntero(v) = res)))}
Complejidad: $O(|valor|)$
Descripción: Dados una tabla t y un valor v, devuelve un puntero al registro perteneciente a la tabla cuyo valor de

su campo clave es v.

ValorEnTabla : Tabla × Valor $\longrightarrow$ bool

ValorEnTabla(t, v) $\equiv$ $(\exists r : Registro)$(r $\in$ registros(t) $\wedge$ $(\exists nc : NombreCampo)$(nc = clave(t)) $\wedge$ (r[nc] = v))

Representación

Descripcion de representacion de tabla

`Tabla` se representa con `estr`

donde `estr` es `tupla`(*camposXClave*: `DiccString(valorClave: string, reg: itConj)`,
registros: `conj(registro)`,
idPorCampo: `conj(Tupla<idCampo: nat, nc: string>)`,
campoClave: `string`)

Rep

- Todo idCampo que pertenece a alguna tupla perteneciente a e.idPorcampo tiene que ser corresponderse lexicograficamente al nc de la misma.
- #claves(e.camposXClave) = (e.registros)
- #(e.registros) = #(e.idPorCampo)
- $(\forall r : registro)(r \in e.registros \Rightarrow_{\text{L}} campoClave \in claves(r))$
- $(\exists t : tupla < nat, string >)(t \in e.idPorCampo \wedge_{\text{L}} campoClave = t.nc)$
- $(\forall vc : string)(vc \in claves(e.camposXClave) \iff (\forall r : registro)(r \in e.registros \Rightarrow_{\text{L}} (r[e.campoClave] = vc)))$
- $(\forall t : tupla < nat, string >)(t \in e.idPorCampo \iff (\forall r : registro)(r \in e.registros \Rightarrow_{\text{L}} (t.nc \in claves(r)))$
- $(\forall vc : string)(vc \in claves(e.camposXClave) \iff *obtener(e.obtener(e.camposXClave, vc) \in e.registros))$
- $(\forall r1 : registro)(r1 \in e.registros \Rightarrow_{\text{L}} (\forall r2 : registro)((r2 \in e.registros \wedge_{\text{L}} \neg(r1 = r2)) \Rightarrow_{\text{L}} campos(r1) = campos(r2)))$

Abs

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

Abs(e) $\equiv$ t: tabla | campos(t) $=_{\text{obs}}$ campos(dameUno(e.registros))
clave(t) $=_{\text{obs}}$ e.campoDlave
registros(t) $=_{\text{obs}}$ e.registros

Algoritmos

---

iNuevaTabla(**in** c: `conj(nombreCampo)`, **in** *clave*: `nombreCampo`) $\rightarrow$ *res* : tabla

1: *res* $\leftarrow$ $\langle$ diccString::Vacio(), conjunto::Vacio(), conjunto::Vacio(), clave $\rangle$

Complejidad: $O(||)$

---

iInsertar(**in/out** t: `Tabla` **in** r: `Registro`)

if Pertenece?(e.registros,r) **then**
  eliminar(e.registros,r)
  borrar(e.camposXClave, r[e.campoXClave])
end if

---

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

if (definido?(camposPorClave, v)) **then**
  borrar(camposXClave, v) //$O(|nombreCampo|)$
  eliminar(registros, v) //$O(|nombreCampo|)$
end if

---

iEsClaveTabla(**in** t: `tabla`, **in** nc: `string`) $\rightarrow$ *res* : ?

return tabla::clave(t) == nombreCampo //$O(|nombreCampo|)$

---

iMergeReg(**in** $r1$: `registro`, **in** $r2$: `registro`) $\rightarrow$ res : ?

```
regRes ← Nuevo()                                                    //O(1)
itReg ← itConj(Campos(r2))                                          //O(1)
while haySiguiente(itReg) do                                        //O(1)
    registro::definir(r1, siguiente(itReg), r2[siguiente(itReg)])   //O(copy(campo) + O(1)
end while
return r1                                                           //O(1)
```

iCampos(**in** t : `tabla` $\rightarrow$ res : conj(nombreCampo)

```
res ← conjunto::vacio()
campos ← idPorCampo
while (conj::haySiguiente(itCampos)) do
    conj::agregar(res, conj::siguiente(itCampos).nc)
    conj::avanzar(itCampos)
end while
return res
```

iRegistros(**in** t : `tabla`) $\rightarrow$ res : conj(Registros)

1: $res \leftarrow registros$

Complejidad :O(1)

iClave(**in** t : `tabla`) $\rightarrow$ res : conj(nombreCampo)

1: $res \leftarrow campoPorClave$

Complejidad :O(1)

iCamposPorClave(**in** t : `tabla` **in** v : `string`) $\rightarrow$ res : puntero(Registro)

```
res ← registro::nuevo()
itReg ← diccString::obtener(tabla.camposXClave, v)
if ( thenhaySiguiente(itReg))
    res ← siguiente(itReg) return res
end if
return &res
```

iObtenerNroCampo(**in** t : `tabla`, **in** nc : `string`) $\rightarrow$ res : nat

```
res ← 1
itTuplaIdPorCampo ← crearIt(tabla.idPorCampo)
while haySiguiente(itTuplaIdPorCampo) do
    if siguiente(itTuplaIdPorCampo).nc = nc then
        res ← siguiente(itTuplaIdPorCampo).idCampo
    end if
    avanzar(itTuplaIdPorCampo)
end while
return res
```

5. Módulo Diccionario String(*string*, σ)

El módulo Diccionario String provee un diccionario básico en el que se puede definir, borrar, y testear si una clave está definida en tiempo $O(|k|)$) donde $|k|$ es la longitud de la clave.

Para describir la complejidad de las operaciones, el costo de copiar un $k \in string$ e igualar $k_1, k_2 \in string$ es igual a $O(|k|)$).

Por ultimo vamos a llamar $copy(s)$ al costo de copiar el elemento $s \in \sigma$ y $equal(s_1, s_2)$ al costo de evaluar si dos elementos $s_1, s_2 \in \sigma$ son iguales.

Interfaz

parámetros formales

géneros $string, \sigma$

función $\bullet = \bullet$(**in** k_1: *string*, **in** k_2: *string*) $\rightarrow$ *res* :**función** `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (k_1 = k_2)$}
Complejidad: $O(|k|)$)
Descripción: función de igualdad de *string*

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

función COPIAR(**in** s: σ) $\rightarrow$ *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($string, \sigma$)

géneros: `dicc`($string, \sigma$).

VACÍO() $\rightarrow$ *res* : `dicc`($string, \sigma$)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacio}
Complejidad: $O(1)$
Descripción: genera un diccionario vacío.
Aliasing: no aplica

DEFINIR(**in/out** d: `dicc`($string, \sigma$), **in** k: *string*, **in** s: σ)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(d, k, s)}
Complejidad: $O(|k| + copy(s))$
Descripción: Define la clave k con el significado s en el diccionario. Si es que la clave ya existia se sobrescribe el significado
Aliasing: los elementos k y s se definen por copia.

DEFINIDO?(**in** d: `dicc`($string, \sigma$), **in** k: *string*) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(d, k)}
Complejidad: $O(|k|)$
Descripción: devuelve `true` si y sólo k está definido en el diccionario.
Aliasing: no aplica

OBTENER(**in** d: `dicc`($string, \sigma$), **in** k: *string*) $\rightarrow$ *res* : σ
Pre $\equiv$ {def?(d, k)}
Post $\equiv$ {alias($res =_{\text{obs}}$ obtener(d, k))}
Complejidad: $O(|k|)$
Descripción: devuelve el significado de la clave k en d.
Aliasing: *res* es modificable si y sólo si d es modificable.

BORRAR(**in/out** d: `dicc`($string, \sigma$), **in** k: *string*)
Pre $\equiv$ {$d = d_0 \wedge$ def?(d, k)}
Post $\equiv$ {$d =_{\text{obs}}$ borrar(d_0, k)}
Complejidad: $O(|k|)$
Descripción: elimina la clave k y su significado de d.

#CLAVE(**in** d: `dicc`($string, \sigma$)) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}

Post $\equiv \{res =_{\text{obs}} \#\text{claves}(d)\}$
Complejidad: $O(1)$
Descripción: devuelve la cantidad de claves del diccionario.

CLAVES(**in** d: `dicc(`$string, \sigma$`)`) $\rightarrow$ res : `conj(`$string$`)`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{claves}(d)\}$
Complejidad: $O(n * |k|)$, donde $n = \#claves(d)$
Descripción: devuelve el conjunto de claves del diccionario.

COPIAR(**in** d: `dicc(`$string, \sigma$`)`) $\rightarrow$ res : `dicc(`$string, \sigma$`)`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} d\}$
Complejidad: $O(n * (|k| + copy(\text{obtener}(k, d))))$, $\forall k \in claves(d)$, donde $n = \#claves(d)$
Descripción: genera una copia nueva del diccionario.