Algoritmos y Estructuras de Datos II

Primer Cuatrimestre de 2016

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

Trabajo Práctico 2

Diseño

Grupo 15

Integrante	LU	Correo electrónico
Alliani, Federico	183/15	fedealliani@gmail.com
Almada Canosa, Matías Ezequiel	140/15	matias.almada.canosa@gmail.com
Lancioni, Gian Franco	234/15	glancioni@dc.uba.ar
Raposeiras, Lucas Damián	034/15	lucas.raposeiras@outlook.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. **Módulo Dato** **3**

2. **Módulo DiccionarioString(significado)** **8**

3. **Módulo DiccionarioNat(significado)** **15**

4. **Módulo Tabla** **25**

5. **Módulo Base de Datos** **37**

1. Módulo Dato

Interfaz

se explica con: DATO.

géneros: `dato`.

Operaciones básicas de dato

DATOSTRING(**in** s: `string`) $\rightarrow$ res : `dato`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ datoString(s)}
Complejidad: $O\big(long(s)\big)$
Descripción: genera un dato con el string s.

DATONAT(**in** n: `nat`) $\rightarrow$ res : `dato`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ datoNat(n)}
Complejidad: $O(1)$
Descripción: genera un dato con el nat n.

NAT?(**in** d: `dato`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ nat?(d)}
Complejidad: $O(1)$
Descripción: devuelve `true` si el dato ingresado es del tipo nat.

VALORNAT(**in** d: `dato`) $\rightarrow$ res : `nat`
Pre $\equiv$ {nat?(d)}
Post $\equiv$ {$res =_{\text{obs}}$ valorNat(d)}
Complejidad: $O(1)$
Descripción: devuelve el valor del nat del dato d.

VALORSTR(**in** d: `dato`) $\rightarrow$ res : `string`
Pre $\equiv$ {$\neg$nat?(d)}
Post $\equiv$ {alias$\big(res =_{\text{obs}}$ valorStr(d)$\big)$}
Complejidad: $O(1)$
Descripción: devuelve el valor del string del dato d.
Aliasing: devuelve el string por referencia. res no es modificable.

MISMOTIPO?(**in** d_1 : `dato`, **in** d_2 : `dato`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $\big($nat?(d_1) $=_{\text{obs}}$ nat?(d_2)$\big)$}
Complejidad: $O(1)$
Descripción: devuelve `true` si d_1 y d_2 son del mismo tipo.

STRING?(**in** d: `dato`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $\neg$nat?(d)}
Complejidad: $O(1)$
Descripción: devuelve `true` si el dato ingresado es del tipo string.

MIN(**in** cs: `conj(dato)`) $\rightarrow$ res : `dato`
Pre $\equiv$ {$\neg$vacía?(s) $\wedge$ ($\forall$ d_1, d_2 : dato) $\Big(\big($está?(d_1, s) $\wedge$ está?(d_2, s)$\big)$ $\Rightarrow$ mismoTipo?(d_1, d_2)$\Big)$}
Post $\equiv$ {$res =_{\text{obs}}$ min(cs)}
Complejidad: $O(\#cs * L)$, donde L es la longitud del string más largo.
Descripción: devuelve el mínimo del conjunto de datos.
Aliasing: res no es modificable

MAX(**in** cs: `conj(dato)`) $\rightarrow$ res : `dato`
Pre $\equiv$ {$\neg$vacía?(s) $\wedge$ ($\forall$ d_1, d_2 : dato) $\Big($(está?(d_1, s) $\wedge$ está?(d_2, s)) $\Rightarrow$ mismoTipo?(d_1, d_2)$\Big)$}
Post $\equiv$ {$res =_{\text{obs}} \max(cs)$}
Complejidad: $O(\#cs * L)$, donde L es la longitud del string más largo.
Descripción: devuelve el máximo del conjunto de datos.
Aliasing: res no es modificable

• $\leq$ • (**in** d_1 : `dato`, **in** d_2 : `dato`) $\rightarrow$ res : `bool`
Pre $\equiv$ {mismoTipo?(d_1, d_2)}
Post $\equiv$ {$\Big($nat?(d_1) $\Rightarrow$ $(res =_{\text{obs}} (\text{valorNat}(d_1) \leq_{nat} \text{valorNat}(d_2)))\Big) \wedge_{\text{L}}$
$\Big($string?(d_1) $\Rightarrow$ $(res =_{\text{obs}} (\text{valorStr}(d_1) \leq_{string} \text{valorStr}(d_2)))\Big)$}
Complejidad: $O\big(min\{|valorStr(d_1)|, |valorStr(d_2)|\}\big)$
Descripción: devuelve `true` si $d_1 \leq d_2$.

• = • (**in** d_1 : `dato`, **in** d_2 : `dato`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$\Big($nat?(d_1) $\Rightarrow$ $(res =_{\text{obs}} (\text{valorNat}(d_1) =_{\text{obs}} \text{valorNat}(d_2)))\Big) \wedge_{\text{L}}$
$\Big($string?(d_1) $\Rightarrow$ $(res =_{\text{obs}} (\text{valorStr}(d_1) =_{\text{obs}} \text{valorStr}(d_2)))\Big)$}
Complejidad: $O\big(min\{|valorStr(d_1)|, |valorStr(d_2)|\}\big)$
Descripción: devuelve `true` si $d_1 = d_2$.

COPIAR(**in** d: `dato`) $\rightarrow$ res : `dato`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} d$}
Complejidad: nat?(d) $\Rightarrow$ $O(1)$
$\neg$nat?(d) $\Rightarrow$ $O(|valorStr(d)|)$
Descripción: devuelve una copia del elemento d.

Representación

Representación de dato

`dato` **se representa con** `estrDato`

donde `estrDato` es `tupla`(*nat?*: `bool`, *valorSrt*: `string`, *valorNat*: `nat`)

Rep : estrDato $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff \Big($(e.nat? $\Rightarrow$ (e.valorStr $=_{\text{obs}}$ "vacio")) $\wedge$ ($\neg e$.nat? $\Rightarrow$ (e.valorNat $=_{\text{obs}}$ 0))$\Big)$

Justificación: Si bien podríamos ser menos restrictivos, esto nos va a permitir considerar el costo de copiar datos nat como $O(1)$

Abs : estrDato e $\longrightarrow$ dato {Rep(e)}
Abs(e) $=_{\text{obs}}$ d: dato | Nat?(d) $=_{\text{obs}}$ e.nat? $\wedge_{\text{L}}$ (e.nat? $\Rightarrow$ (valorNat(d) $=_{\text{obs}}$ e.valorNat) $\wedge$
$\neg$(e.nat?) $\Rightarrow$ (valorStr(d) $=_{\text{obs}}$ e.valorStr))

Algoritmos

Algoritmos de dato

---

iDatoString(**in** s : `string`) $\rightarrow$ res : estrDato

$res.nat? \leftarrow false$ ▷ $O(1)$

$res.valorStr \leftarrow copiar(s)$ /* complejidad heredada de Copiar de Vector(α) */ ▷ $O\left(\sum_{i=1}^{long(s)} copy(s[i])\right)$

$res.valorNat \leftarrow 0$ ▷ $O(1)$

Complejidad: $O\big(long(s)\big)$

Justificación: $O(1) + O\left(\sum_{i=1}^{long(s)} copy(s[i])\right) + O(1) = O\left(\sum_{i=1}^{long(s)} copy(s[i])\right) = \sum_{i=1}^{long(s)} O\big(copy(s[i])\big)$

$= \sum_{i=1}^{long(s)} O(1) = \text{long}(s) * O(1) = O\big(long(s)\big)$

---

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

$res.nat? \leftarrow true$ ▷ $O(1)$

$res.valorStr \leftarrow$ "$vacio$" ▷ $O\big(long(\text{"}vacio\text{"})\big) = O(1)$

$res.valorNat \leftarrow n$ ▷ $O(1)$

Complejidad: $O(1)$

Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

---

iNat?(**in** e : `estrDato`) $\rightarrow$ res : bool

$res \leftarrow e.nat?$ ▷ $O(1)$

Complejidad: $O(1)$

Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

---

iValorNat(**in** e : `estrDato`) $\rightarrow$ res : nat

$res \leftarrow e.valorNat$ ▷ $O(1)$

Complejidad: $O(1)$

Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

---

iValorStr(**in** e : `estrDato`) $\rightarrow$ res : string

$res \leftarrow e.valorStr$ ▷ $O(1)$

Complejidad: $O(1)$

Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

---

iMismoTipo?(**in** e_1 : `estrDato`, **in** e_2 : `estrDato`) $\rightarrow$ res : bool

$res \leftarrow (e_1.nat? = e_2.nat?)$ ▷ $O(1)$

Complejidad: $O(1)$

Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

---

iString?(**in** e: `estrDato`) → res : bool

$res \leftarrow$ **not** $e.nat?$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

iMin(**in** cs: `conj(estrDato)`) → res : estrDato

$it \leftarrow CrearIt(cs)$ ▷ $O(1)$
$res \leftarrow Siguiente(it)$ ▷ $O(1)$
while $HaySiguiente(it)$ **do** ▷ $O(\#cs * ...)$
// complejidad heredada de $\bullet \leq_i \bullet$
  if not $res \leq_i Siguiente(it)$ **then** ▷ $O\big(min\{|res.valorStr|, |Siguiente(it).valorStr|\}\big)$
    $res \leftarrow Siguiente(it)$ ▷ $O(1)$
  end if
  $Avanzar(it)$ ▷ $O(1)$
end while

Complejidad: $O(\#cs * L)$, donde L es la longitud del string más largo
Justificación: Se recorre toda la lista $\big(O(\#cs)\big)$ y cada elemento se compara con el auxiliar res. En algún momento, res se va a comparar con el string de longitud L.

iMax(**in** cs: `conj(estrDato)`) → res : estrDato

$it \leftarrow CrearIt(cs)$ ▷ $O(1)$
$res \leftarrow Siguiente(it)$ ▷ $O(1)$
while $HaySiguiente(it)$ **do** ▷ $O(\#cs * ...)$
// complejidad heredada de $\bullet \leq_i \bullet$
  if $res \leq_i Siguiente(it)$ **then** ▷ $O\big(min\{|res.valorStr|, |Siguiente(it).valorStr|\}\big)$
    $res \leftarrow Siguiente(it)$ ▷ $O(1)$
  end if
  $Avanzar(it)$ ▷ $O(1)$
end while

Complejidad: $O(\#cs * L)$, donde L es la longitud del string más largo
Justificación: Se recorre toda la lista $\big(O(\#cs)\big)$ y cada elemento se compara con el auxiliar res. En algún momento, res se va a comparar con el string de longitud L.

$\bullet \leq_i \bullet$ (**in** e_1: `estrDato`, **in** e_2: `estrDato`) → res : bool

if $e_1.nat?$ **then** ▷ $O(1)$
  $res \leftarrow (e_1.valorNat \leq e_2.valorNat)$ ▷ $O(1)$
else
  $res \leftarrow true$ ▷ $O(1)$
  while res **and** $i < min\big(longitud(e_1.valorStr), longitud(e_2.valorStr)\big)$ **do** ▷ $O(1)$
    if $e_1.valorStr[i] > e_2.valorStr[i]$ **then** ▷ $O(1)$
      $res \leftarrow false$ ▷ $O(1)$
    end if
  end while
  $res \leftarrow (e_1.valorStr \leq e_2.valorStr)$ ▷ $O\big(min\{|e_1.valorStr|, |e_2.valorStr|\}\big)$
end if

Complejidad: $O\big(min\{|e_1.valorStr|, |e_2.valorStr|\}\big)$
Justificación: Para determinar la desigualdad entre ambos vectores de strings realiza comparaciones entre chars $O(1)$ hasta el mínimo de las longitudes

---

$\bullet = \bullet$ (**in** e_1 : `estrDato`, **in** e_2 : `estrDato`) $\rightarrow$ res : bool

if $MismoTipo?(e_1, e_2)$ **then** ▷ $O(1)$
  if $e_1.nat?$ **then** ▷ $O(1)$
    $res \leftarrow (e_1.valorNat = e_2.valorNat)$ ▷ $O(1)$
  else
    $res \leftarrow (e_1.valorStr = e_2.valorStr)$ ▷ $O\big(min\{|e_1.valorStr|, |e_2.valorStr|\}\big)$
  end if
else
  $res \leftarrow false$ ▷ $O(1)$
end if

Complejidad: $O\big(min\{|e_1.valorStr|, |e_2.valorStr|\}\big)$
Justificación: $O(1) + O(1) + O(1) + O\big(min\{|e_1.valorStr|, |e_2.valorStr|\}\big) = O\big(min\{|e_1.valorStr|, |e_2.valorStr|\}\big)$

---

iCopiar(**in** e : `estrDato`) $\rightarrow$ res : estrDato

$res \leftarrow \langle e.nat?, e.valorStr, e.valorNat \rangle$ ▷ $O(1) + O(|e.valorStr|) + O(1)$

Complejidad: $O(|e.valorStr|)$
Justificación: La complejidad del algoritmo es la complejidad de copiar un string. La complejidad de copiar un string es $O\big(long(string)\big)$. Si esNat? es true, por invariante, el largo de valorStr está acotado (valorStr = "vacio") y la complejidad es $O(1)$

---

2. Módulo DiccionarioString(significado)

Interfaz

parámetros formales
géneros significado
función COPIAR(**in** *sig*: `significado`) → *res* : `significado`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} sig$}
Complejidad: $O(copy(sig))$
Descripción: vuelve una copia del parámetro.

se explica con: DICCIONARIO EXTENDIDO(STRING, SIGNIFICADO).

géneros: `diccString(significado)`.

Operaciones básicas de diccString

VACÍO() → *res* : `diccString(significado)`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ vacio}
Complejidad: $O(1)$
Descripción: genera un DiccionarioString vacío.

DEFINIR(**in** *clave*: `string`, **in** *significado*: `significado`, **in/out** *dicc*: `diccString(significado)`)
Pre ≡ {¬def?(*clave*,*dicc*) ∧ $dicc_0 = dicc$}
Post ≡ {$dicc =_{\text{obs}}$ definir(*clave*, *significado*, $dicc_0$)}
Complejidad: $O(max\{long(clave), copy(significado)\})$
Descripción: define la clave ingresada en el diccionario.

DEF?(**in** *clave*: `string`, **in** *dicc*: `diccString(significado)`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ def?(*clave*, *dicc*)}
Complejidad: $O(long(clave))$
Descripción: devuelve `true` si la clave está definida en el diccionario.

OBTENER(**in** *clave*: `string`, **in** *dicc*: `diccString(significado)`) → *res* : `significado`
Pre ≡ {def?(*clave*, *dicc*)}
Post ≡ {alias($res =_{\text{obs}}$ obtener(*clave*, *dicc*))}
Complejidad: $O(long(clave))$
Descripción: devuelve el significado correspondiente a la clave ingresada.
Aliasing: se genera alias entre *res* y el significado en el diccionario si el tipo significado no es primitivo. *res* no es modificable.

CLAVES(**in** *dicc*: `diccString(significado)`) → *res* : `itConjunto(string)`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ claves(*dicc*)}
Complejidad: $O(\#claves(dicc) * M)$, donde M es la longitud del mayor string clave.
Descripción: devuelve un iterador al conjunto de las claves del diccionario ingresado.

BORRAR(**in** *clave*: `string`, **in/out** *dicc*: `diccString(significado)`)
Pre ≡ {def?(*clave*, *dicc*) ∧ $dicc_0 = dicc$}
Post ≡ {$dicc =_{\text{obs}}$ borrar(*clave*, $dicc_0$)}
Complejidad: $O(long(clave))$
Descripción: borra la clave del diccionario.

VISTADICC(**in/out** *dicc*: `diccString(significado)`) → *res* : `itBi(tupla( Clave: string, significado: significado))`
Pre ≡ {true}
Post ≡ {alias($dicc =_{\text{obs}}$ secuADicc(secuSuby(*res*)))}

Complejidad: $O(1)$
Descripción: devuelve un iterador a una lista de tuplas con las claves y sus significados.
Aliasing: el iterador no es modificable.

COPIAR(**in** *dicc*: `diccString(significado)`) → *res* : `diccString(significado)`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} dicc$}
Complejidad: $O\big(\#claves(dicc) * max\{k, s\}\big)$, donde k es la longitud máxima de cualquier clave en *dicc* y s el máximo costo de copiar un significado de *dicc* de dicho tipo.
Descripción: devuelve una copia sin aliasing del diccionario de entrada.

MIN(**in** *dicc*: `diccString(significado)`) → *res* : `string`
Pre ≡ {#claves(e) >0}
Post ≡ {alias($res =_{\text{obs}} m$) | $\big(\forall\ n \in \text{claves}(d)\big)\ (m \leq n)$}
Complejidad: $O\big(long(|min(claves(dicc))|)\big)$
Descripción: Devuelve la clave mínima del diccionario. La clave mínima es utilizando el orden de la funcion ORD y no utilizando orden lexicografico.

MAX(**in** *dicc*: `diccString(significado)`) → *res* : `string`
Pre ≡ {#claves(e) >0}
Post ≡ {alias($res =_{\text{obs}} m$) | $\big(\forall\ n \in \text{claves}(d)\big)\ (m \geq n)$}
Complejidad: $O\big(long(|min(claves(dicc))|)\big)$
Descripción: Devuelve la clave máxima del diccionario. La clave máxima es utilizando el orden de la funcion ORD y no utilizando orden lexicografico.

Representación

Representación de diccString

`diccString(significado)` **se representa con** `estrDiccString`

donde `estrDiccString` es `tupla` (*trie*: `nodoTrie`, *valores*: `lista`(`tupla<`*clave*: `string`, *significado*: `significado>`))
donde `nodoTrie` es `tupla`(*valor*: `puntero`(`itLista(tupla<string, significado>)`), *hijos*: `arreglo[256] de puntero(nodoTrie)`, *cantHijos*: `nat`)

Rep : estrDiccString ⟶ bool
Rep(e) ≡ true ⟺
1) Un valor está en la lista si y solo si hay un nodo en e.trie que apunta a un iterador cuyo siguiente es ese valor.
2) La clave de ese valor corresponde al string formado concatenando los valores char del índice de cada hijo que se recorre, dicho recorrido es único (p.e: "*Alas*" solamente está definido si el nodo correspondiente al recorrido A→L→A→S apunta a un valor no nulo). Por lo tanto, el primer nodo no puede apuntar a un valor válido.
3) cantHijos es igual a la cantidad de punteros no nulos en hijos.
4) No existen dos nodos en la estructura recursiva que compartan alguno de sus hijos.

Abs : estrDiccString e ⟶ diccString(significado) {Rep(e)}
Abs(e) =obs d: diccString(significado) | (∀ c : string) def?(c, d) ⟺ esClave?(c, e.valores) ∧L obtener(c, d) =obs significado(c, e.valores)
esClave? : string s × secu(tupla(string,α)) xs ⟶ bool {Rep(e)}
esClave?(s, xs) ≡ ¬vacia?(xs) ∧L $\big(\Pi_1(\text{prim}(xs))\ =_{\text{obs}}\ s \vee \text{esClave?}(s, \text{fin}(xs))\big)$

significado : string s × secu(tupla(string,α)) xs ⟶ α {Rep(e) ∧L esClave?(s, xs)}
significado(s, xs) ≡ **if** $\Pi_1(\text{prim}(xs)) =_{\text{obs}} s$ **then** $\Pi_2(\text{prim}(xs))$ **else** significado$\big(s, \text{fin}(xs)\big)$ **fi**

Algoritmos

Algoritmos de diccString

iVacio() → *res* : estrDiccString

res.valores ← $Vacia()$ ▷ $O(1)$
res.trie ← $iNuevoNodo()$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

---

iNuevoNodo() → *res* : nodoTrie // no se exporta

res ← $\langle NULL, CrearArreglo(256), 0 \rangle$ ▷ $O(1)$
for $i \leftarrow 0$ **to** 255 **do** ▷ $O(255 * ...) = O(1 * ...)$
  res.hijos[i] ← $NULL$ ▷ $O(1)$
end for

Complejidad: $O(1)$
Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

---

iDefinir(**in** *clave*: `string`, **in** *significado*: `significado`, **in/out** *e*: `estrDiccString`)

entrada ← $\langle clave, significado \rangle$ ▷ $O(1)$

// agregamos a la lista
iter ← $AgregarAdelante(e.valores, entrada)$ ▷ $O\big(copy(entrada)\big) = O\big(copy(clave)\big) + O\big(copy(significado)\big)$
// iter tiene a entrada como siguiente

actual ← &*e.trie* // actual es de tipo puntero(nodoTrie) ▷ $O(1)$
for $i \leftarrow 0$ **to** $Longitud(clave) - 1$ **do** ▷ $O\big(long(clave) * ...\big)$
  if $\big(actual \rightarrow hijos\big)\big[ord(clave[i])\big] = NULL$ **then** ▷ $O(1)$
    $\big(actual \rightarrow hijos\big)\big[ord(clave[i])\big] \leftarrow \&\big(iNuevoNodo()\big)$ ▷ $O(1)$
    $(actual \rightarrow cantHijos) \leftarrow (actual \rightarrow cantHijos) + 1$ ▷ $O(1)$
  end if
  actual ← $\big(actual \rightarrow hijos\big)\big[ord(clave[i])\big]$ ▷ $O(1)$
end for
$(actual \rightarrow valor) \leftarrow \&iter$ ▷ $O(1)$

Complejidad: $O\big(Longitud(clave)\big) + O\big(copy(clave)\big) + O\big(copy(significado)\big)$
$= O\big(max\{Longitud(clave), copy(significado)\}\big)$
Justificación:

Para definir creamos una tupla y copiamos la clave y el significado, por lo tanto tenemos en complejidad la copia del más grande de los dos, es decir, $O\big(copy(clave)\big) + O\big(copy(significado)\big) = O(L) + O\big(copy(significado)\big)$.

Luego se hace un ciclo que se realiza L veces y hace operaciones $O(1)$.

Por lo tanto la complejidad queda $O(L) + O\big(copy(significado)\big) + O(L)$. Por ser sumas queda la mayor de ellas, es decir $max\{2 * O(L), O\big(copy(significado)\big)\} = O\big(max\{L, copy(significado)\}\big)$

L: Longitud de la clave.

iDef?(**in** *clave*: `string`, **in** *e*: `estrDiccString`) → *res* : bool

```
actual ← &e.trie                                                    ▷ O(1)
res ← true                                                          ▷ O(1)
i ← 0                                                               ▷ O(1)
while i < Longitud(clave) and res do                                ▷ O(long(clave) ∗ ...)
    if actual → cantHijos > 0 then                                  ▷ O(1)
        if (actual → hijos)[ord(clave[i])] = NULL then              ▷ O(1)
            res ← false                                             ▷ O(1)
        else
            actual ← (actual → hijos[ord(clave[i])])                ▷ O(1)
        end if
        i++                                                         ▷ O(1)
    else
        res ← false                                                 ▷ O(1)
    end if
end while

if res then                                                         ▷ O(1)
    res ← not ((actual → valor) = NULL)                             ▷ O(1)
end if
```

Complejidad: $O(long(clave))$
Justificación: El ciclo itera a lo sumo long($clave$) veces.

---

iObtener(**in** *clave*: `string`, **in** *e*: `estrDiccString`) → *res* : significado

```
actual ← &e.trie                                                    ▷ O(1)
for i ← 0 to Longitud(clave) − 1 do                                 ▷ O(long(clave) ∗ ...)
    actual ← (actual → hijos[ord(clave[i])])                        ▷ O(1)
end for
res ← Siguiente(∗(actual → valor)).significado                      ▷ O(1)
```

Complejidad: $O(long(clave))$
Justificación: El ciclo itera a lo sumo long($clave$) veces.

---

iClaves(**in** *e*: `estrDiccString`) → *res* : itConj(string)

```
itLista ← CrearIt(e.valores)                                        ▷ O(1)
aux ← Vacio() // aux : conj(string)                                 ▷ O(1)
while HaySiguiente?(it) do                                          ▷ O(long(e.valores) ∗ ...)
    res ← AgregarRapido(aux, Siguiente(it).clave)                   ▷ O(copy(clave))
    Avanzar(it)                                                     ▷ O(1)
end while
```

Complejidad: $O(long(e.valores) * M)$, donde M es la longitud del mayor string clave en e.valores.
Justificación: El ciclo itera toda la lista copiando la clave de cada elemento. Se acota el costo de copiado de todos los elementos por el del copiado de mayor longitud.

---

iBorrar(**in** *clave*: `string`, **in/out** *e*: `estrDiccString`)

```
actual ← &e.trie                                                          ▷ O(1)
listo ← false                                                             ▷ O(1)
for i ← 0 to Longitud(clave) − 1 do                                       ▷ O(long(clave) ∗ ...)
    temp ← (actual → hijos[ord(clave[i])]) // guardamos a dónde apunta    ▷ O(1)
    if (actual → hijos[ord(clave[i])]) → cantHijos = 0 then               ▷ O(1)
        (actual → hijos[ord(clave[i])]) ← NULL                            ▷ O(1)
    end if
    actual ← temp                                                         ▷ O(1)
    // seguimos recorriendo lo que antes era su nodo hijo para liberar el resto de memoria
end for
EliminarSiguiente(∗(actual → valor))                                      ▷ O(1)

// crea un iterador uniendo la lista antes del elemento más la lista después del elemento
(actual → valor) ← NULL                                                   ▷ O(1)
actual ← &e.trie                                                          ▷ O(1)
i ← 0                                                                     ▷ O(1)
while i < Longitud(clave) − 1and not listo do                             ▷ O(long(clave) ∗ ...)
    if (actual → hijos[ord(clave[i])]) → cantHijos > 0 then               ▷ O(1)
        actual ← (actual → hijos[ord(clave[i])])                          ▷ O(1)
    else
        (actual → hijos[ord(clave[i])]) ← NULL                            ▷ O(1)
        listo ← true                                                      ▷ O(1)
    end if
    i++                                                                   ▷ O(1)
end while
```

Complejidad: $O(long(clave))$
Justificación: Ambos ciclos iteran a lo sumo long($clave$) veces.

---

iVistaDicc(**in** *e*: `estrDiccString`) → *res* : itLista(tupla⟨clave: string, significado: significado⟩)

```
res ← CrearIt(e.valores)                                                  ▷ O(1)
```

Complejidad: $O(1)$
Justificación: El algoritmo tiene una única llamada a una función con costo $O(1)$.

---

iCopiar(**in** *e*: `estrDiccString`) → *res* : estrDiccString

```
it ← CrearIt(e.valores)                                                   ▷ O(1)
res ← Vacio()                                                             ▷ O(1)
while HaySiguiente(it) do                                                 ▷ O(long(e.valores) ∗ ...)
    Definir(Siguiente(it).clave, Siguiente(it).significado, res)          ▷ O(max{K, S})
    Avanzar(it)                                                           ▷ O(1)
end while
```

Complejidad: $O(long(e.valores) * max\{K, S\})$, donde K es la longitud máxima de cualquier clave en e y S el máximo costo de copiar un significado de e de dicho tipo.
Justificación: El ciclo itera toda la lista definiendo cada clave en un nuevo diccionario.

---

iMin(**in** e : `estrDiccString`) $\rightarrow$ res : string

```
actual ← &e.trie                                                          ▷ O(1)
termine ← false                                                           ▷ O(1)
while not termine do                                                      ▷ O(L * ...)
    if (actual → valor) = NULL then                                       ▷ O(1)
        for i ← 0 to 255 do                                               ▷ O(255 * ...) = O(1 * ...)
            if not actual → hijos[ord(clave[i])] = NULL then              ▷ O(1)
                actual ← (actual → hijos[ord(clave[i])])                  ▷ O(1)
            end if
        end for
    else
        termine ← true                                                    ▷ O(1)
        res ← Siguiente(*(actual → valor)).clave                          ▷ O(1)
    end if
end while
```

Complejidad: $O\big(long\big(|min(claves(e))|\big)\big)$

Justificación:

En el algoritmo hay un ciclo principal (while) y un ciclo interno (for) y luego fuera de los ciclos operaciones $O(1)$. El ciclo del while itera hasta que encuentra la primer palabra completa recorriendo desde el nodo del trie buscando siempre desde el primer char hasta el último.

Entonces iteramos L veces, por lo tanto tenemos $O(L)$. Pero dentro del while hay un ciclo interno(for) que itera siempre 255 veces, por lo tanto tenemos de complejidad $O(L * 255)$. La constante se puede sacar y queda $O(L)$.

Todas las operaciones internas de los ciclos son $O(1)$.

L: Longitud de la clave mínima del diccionario.

```
iMax(in e: estrDiccString) → res : string
  actual ← &e.trie                                                    ▷ O(1)
  termine ← false                                                     ▷ O(1)
  while not termine do                                                ▷ O(L * ...)
    // Quiero que mientras haya hijos se meta en el índice más grande
    if (actual → cantHijos) = 0 then                                  ▷ O(1)
      res ← Siguiente(*(actual → valor)).clave                        ▷ O(1)
    else
      i ← 255                                                         ▷ O(1)
      seguir ← true                                                   ▷ O(1)

      while i ≥ 0 and seguir do
        if not actual → hijos[ord(clave[i])] = NULL then              ▷ O(1)
          actual ← (actual → hijos[ord(clave[i])])                    ▷ O(1)
          seguir ← false                                              ▷ O(1)
        end if
        i − −                                                         ▷ O(1)
      end while
    end if
  end while
```

Complejidad: $O(long(|min(claves(e))|))$

Justificación:

En el algoritmo hay un ciclo principal (while) y un ciclo interno (for) y luego fuera de los ciclos operaciones $O(1)$.El ciclo del while itera hasta que encuentra la primer palabra completa recorriendo desde el nodo del trie buscando siempre desde el último char hasta el primero.

Entonces iteramos L veces por lo tanto tenemos $O(L)$. Pero dentro del while hay un ciclo interno(for) que itera siempre 255 veces, por lo tanto tenemos de complejidad $O(L * 255)$. La constante se puede sacar y queda $O(L)$.

Todas las operaciones internas de los ciclos son $O(1)$.

L: Longitud de la clave máxima del diccionario.

3. Módulo DiccionarioNat(significado)

Interfaz

parámetros formales
géneros significado
función COPIAR(**in** *sig*: `significado`) → *res* : `significado`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} sig$}
Complejidad: $O(copy(sig))$
Descripción: vuelve una copia del parámetro.
se explica con: DICCIONARIO EXTENDIDO(NAT, SIGNIFICADO), ITERADOR UNIDIRECCIONAL(TUPLA(NAT, SIGNIFICADO)).

géneros: `diccNat(significado)`, `itDiccNat(significado)`.

Operaciones básicas de diccNat

VACÍO() → *res* : `diccNat(significado)`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ vacio}
Complejidad: $O(1)$
Descripción: genera un DiccionarioNat vacío.

DEFINIR(**in** *clave*: `nat`, **in** *significado*: `significado`, **in/out** *dicc*: `diccNat(significado)`)
Pre ≡ {¬def?(*clave*,*dicc*) ∧ $dicc_0 = dicc$}
Post ≡ {$dicc =_{\text{obs}}$ definir(*clave*, *significado*, $dicc_0$)}
Complejidad: $O(\#claves(dicc))$ / $O(log\ \#claves(dicc))$ asumiendo distribución uniforme de claves.
Descripción: define la clave ingresada en el diccionario.

DEF?(**in** *clave*: `nat`, **in** *dicc*: `diccNat(significado)`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ def?(*clave*, *dicc*)}
Complejidad: $O(\#claves(dicc))$ / $O(log\ \#claves(dicc))$ asumiendo distribución uniforme de claves.
Descripción: devuelve `true` si la clave está definida en el diccionario.

OBTENER(**in** *clave*: `nat`, **in** *dicc*: `diccNat(significado)`) → *res* : `significado`
Pre ≡ {def?(*clave*, *dicc*)}
Post ≡ {alias($res =_{\text{obs}}$ obtener(*clave*, *dicc*))}
Complejidad: $O(\#claves(dicc))$ / $O(log\ \#claves(dicc))$ asumiendo distribución uniforme de claves.
Descripción: devuelve el significado correspondiente a la clave ingresada.
Aliasing: se genera alias entre *res* y el significado en el diccionario si el tipo significado no es primitivo. *res* no es modificable.

BORRAR(**in** *clave*: `nat`, **in/out** *dicc*: `diccNat(significado)`)
Pre ≡ {def?(*clave*, *dicc*) ∧ $dicc_0 = dicc$}
Post ≡ {$dicc =_{\text{obs}}$ borrar(*clave*, $dicc_0$)}
Complejidad: $O(\#claves(dicc))$ / $O(log\ \#claves(dicc))$ asumiendo distribución uniforme de claves.
Descripción: borra la clave del diccionario.

MIN(**in** *dicc*: `diccNat(significado)`) → *res* : `tupla(nat, significado)`
Pre ≡ {#claves(*dicc*) >0}
Post ≡ {alias($\Pi_1(res) =_{\text{obs}} \min(\text{claves}(dicc))$) ∧$_{\text{L}}$ alias($\Pi_2(res) =_{\text{obs}}$ obtener($\Pi_1(res)$, *dicc*))}
Complejidad: $O(\#claves(dicc))$ / $O(log\ \#claves(dicc))$ asumiendo distribución uniforme de claves.
Descripción: devuelve una tupla con la clave mínima y su significado.
Aliasing: *res* no es modificable.

MAX(**in** *dicc*: `diccNat(significado)`) → *res* : `tupla(nat, significado)`
Pre ≡ {#claves(*dicc*) >0}
Post ≡ {alias($\Pi_1(res) =_{\text{obs}} \max(\text{claves}(dicc))$) ∧$_{\text{L}}$ alias($\Pi_2(res) =_{\text{obs}}$ obtener($\Pi_1(res)$, *dicc*))}

Complejidad: $O(\#claves(dicc))$ / $O(log\ \#claves(dicc))$ asumiendo distribución uniforme de claves.
Descripción: devuelve una tupla con la clave máxima y su significado.
Aliasing: *res* no es modificable.

Operaciones básicas del iterador

CrearIt(**in** *dicc*: `diccNat(significado)`) → *res* : `itDiccNat(significado)`
Pre ≡ {true}
Post ≡ {alias(secuADicc(Siguientes(*res*)) =obs *dicc*)}
Complejidad: $O(1)$
Descripción: crea un iterador del conjunto.
Aliasing: el iterador no puede realizar modificaciones y se indefine con la inserción y eliminación de elementos en el diccionario.

Siguientes(**in** *it*: `itDiccNat(significado)`) → *res* : `lista(tupla(nat, significado))`
Pre ≡ {true}
Post ≡ {*res* =obs siguientes(*it*)}
Complejidad: $O(n * copy(x))$, siendo n la cantidad de claves del diccionario y x el significado mas costoso de copiar.
Descripción: devuelve una lista con las claves siguientes y sus significados.
Aliasing: no hay ya que se copian los elementos.

Avanzar(**in/out** *it*: `itDiccNat(significado)`)
Pre ≡ {HayMas?(*it*) ∧ it_0 = *it*}
Post ≡ {*it* =obs Avanzar(it_0)}
Complejidad: $O(1)$
Descripción: avanza a la posición siguiente del iterador.

HayMas?(**in** *it*: `itDiccNat(significado)`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {*res* =obs HayMas?(*it*)}
Complejidad: $O(1)$
Descripción: devuelve `true` si y sólo si el en el iterador todavía quedan elementos para avanzar.

Actual(**in** *it*: `itDiccNat(significado)`) → *res* : `tupla(nat, significado)`
Pre ≡ {HayMas?(*it*)}
Post ≡ {alias(*res* =obs Actual(*it*))}
Complejidad: $O(1)$
Descripción: devuelve el elemento correspondiente a la posición actual del iterador.
Aliasing: Genera aliasing. *res* no es modificable.

Representación

Representación de diccNat

`diccNat(significado)` **se representa con** `estrDiccNat`

donde `estrDiccNat` es `puntero(nodoDiccNat(significado))`
donde `nodoDiccNat(significado)` es `tupla`(*clave*: `nat`, *significado*: `puntero(significado)`, *izq*: `puntero(nodoDiccNat)`, *der*: `puntero(nodoDiccNat)`)

Invariante de representación
1) Hijo izq menor estricto e hijo derecho mayor estricto.
2) Rep recursivo en sus hijos.
3) El significado no es nulo.
4) No hay repetidos.

Rep : estrDiccNat ⟶ bool

$\text{Rep}(e) \equiv \text{true} \iff (e =_{\text{obs}} \text{NULL}) \vee_{\text{L}} \big((e \to \text{izq} \neq_{\text{obs}} \text{NULL}) \Rightarrow (e \to \text{izq} \to \text{clave}) < e \to \text{clave}\big)$
$\wedge \big((e \to \text{der} \neq_{\text{obs}} \text{NULL}) \Rightarrow (e \to \text{der} \to \text{clave}) > e \to \text{clave}\big)$
$\wedge \text{Rep}(e \to \text{izq}) \wedge \text{Rep}(e \to \text{der}) \wedge (e \to \text{significado} \neq_{\text{obs}} \text{NULL}) \wedge \big(\forall\, n : \text{nat}, \text{esClave?}(n, e)\big)\ (n =_{\text{obs}} e \to \text{clave}) \Rightarrow \big(\neg\text{esClave?}(n, e \to \text{izq}) \wedge \neg\text{esClave?}(n, e \to \text{der})\big) \wedge \big(e \neq_{\text{obs}} \text{NULL} \wedge_{\text{L}} \text{esClave?}(n, e \to \text{izq})\big) \Rightarrow \big(n \neq_{\text{obs}} e \to \text{clave} \wedge \neg\text{esClave?}(n, e \to \text{der})\big) \wedge \big(e \neq_{\text{obs}} \text{NULL} \wedge_{\text{L}} \text{esClave?}(n, e \to \text{der})\big) \Rightarrow \big(n \neq_{\text{obs}} e \to \text{clave} \wedge \neg\text{esClave?}(n, e \to \text{izq})\big)$

$\text{Abs} : \text{estrDiccNat}\ e \longrightarrow \text{diccNat(significado)} \quad \{\text{Rep}(e)\}$
$\text{Abs}(e) =_{\text{obs}} \text{d: diccNat(significado)} \mid (\forall\, c : \text{nat})\ \big(\text{def?}(c, d) =_{\text{obs}} \text{esClave?}(c, e)\big) \wedge \big(\text{def?}(c, d) \Rightarrow_{\text{L}} \text{obtener}(c, d) =_{\text{obs}} \text{suSignificado}(c, d)\big)$

$\text{esClave?} : \text{nat}\ n \times \text{estrDiccNat}\ e \longrightarrow \text{bool} \quad \{\text{Rep}(e)\}$
$\text{esClave?}(n, e) \equiv (e \neq_{\text{obs}} \text{NULL}) \wedge_{\text{L}} \big((e \to \text{clave} =_{\text{obs}} n) \vee (\text{esClave?}(n, e \to \text{der})) \vee (\text{esClave?}(n, e \to \text{izq}))\big)$

$\text{suSignificado} : \text{nat}\ n \times \text{estrDiccNat}\ e \longrightarrow \text{significado} \quad \{\text{Rep}(e) \wedge_{\text{L}} \text{esClave?}(n, e)\}$
$\text{suSignificado}(n, e) \equiv$ **if** $(e \to \text{clave}) =_{\text{obs}} c$ **then** $*(e \to \text{significado})$ **else** **if** $(e \to \text{clave}) < c$ **then** $\text{suSignificado}(c, e \to \text{izq})$ **else** $\text{suSignificado}(c, e \to \text{der})$ **fi** **fi**

Representación del iterador

`itDiccNat(significado)` **se representa con** `iter`

donde `iter` es `pila(estrDiccNat)`

Invariante de representación

1) Vale rep para cada elemento de la pila.
2) No hay punteros nulos en la pila.
3 La pila está ordenada decrecientemente.
4) No puede haber dos punteros apilados que compartan algún hijo (por lo tanto nunca va a haber hijos apilados para ningún nodo). Tampoco por rep puede pasar que haya loops, por ejemplo: El nodo con clave 15 apunte a su izq al nodo con clave 9 y este a su der al nodo con clave 15 nuevamente.

$\text{Rep} : \text{iter} \longrightarrow \text{bool}$
$\text{Rep}(i) \equiv \text{true} \iff \big(\forall\, n : \text{estrDiccNat}, \text{estaEnPila?}(n, i)\big)\ \text{Rep}(n) \wedge_{\text{L}}$
$\Big(\text{noApilaNulos}(i) \wedge \text{ordenadaDec}(i) \wedge \big(\forall\, n : \text{estrDiccNat}, \text{estaEnPila?}(n, i)\big)\ \neg\big(\exists\, n' : \text{estrDiccNat}, n \neq_{\text{obs}} n' \wedge \text{estaEnPila?}(n', i)\big)\ \big(\exists\, x : \text{nat}, \text{esta?}(x, \text{clavesDe}(n)) \wedge \text{esta?}(x, \text{clavesDe}(n'))\big)\Big)$

$\text{noApilaNulos} : \text{iter} \longrightarrow \text{bool}$
$\text{noApilaNulos}(i) \equiv$ **if** $\text{vacia?}(i)$ **then** true **else** $\text{tope}(i) \neq_{\text{obs}} \text{NULL} \wedge \text{noApilaNulos}\big(\text{desapilar}(i)\big)$ **fi**

$\text{clavesDe} : \text{estrDiccNat} \longrightarrow \text{secu(nat)}$
$\text{clavesDe}(e) \equiv$ **if** $e \neq_{\text{obs}} \text{NULL}$ **then** $\text{clavesDe}(e \to \text{izq}) \,\&\, \big((e \to \text{clave}) \bullet \text{clavesDe}(e \to \text{der})\big)$ **else** $<>$ **fi**

$\text{ordenadaDec} : \text{estrDiccNat} \longrightarrow \text{bool}$
$\text{ordenadaDec}(e) \equiv \big(\text{tamaño}(e) < 2\big) \vee_{\text{L}} \big((\text{tope}(e) \to \text{clave}) < \text{tope}\big(\text{desapilar}(e) \to \text{clave}\big) \wedge \text{ordenada}\big(\text{desapilar}(e)\big)\big)$

$\text{estaEnPila?} : \alpha \times \text{pila}(\alpha) \longrightarrow \text{bool}$
$\text{estaEnPila?}(e, s) \equiv$ **if** $\text{vacia?}(s)$ **then** false **else** $\text{tope}(s) =_{\text{obs}} e \vee \text{estaEnPila?}\big(e, \text{desapilar}(s)\big)$ **fi**

$\text{Abs} : \text{iter}\ i \longrightarrow \text{itDiccNat(significado)} \quad \{\text{Rep}(i)\}$
$\text{Abs}(i) =_{\text{obs}} \text{it: itDiccNat(significado)} \mid \text{Siguientes}(it) =_{\text{obs}} \text{secuDFS}(i)$

$\text{secuDFS} : \text{iter}\ i \longrightarrow \text{secu}\big(\text{tupla(nat, significado)}\big) \quad \{\text{Rep}(i)\}$

$$
\begin{array}{l}
\text{secuDFS}(i) \equiv \textbf{if } \text{vacía?}(i) \textbf{ then} \\
\quad <> \\
\quad \textbf{else} \\
\quad\quad \textbf{if } \text{tope}(i) \rightarrow \text{der} \neq_{\text{obs}} \text{NULL} \wedge \text{tope}(i) \rightarrow \text{izq} \neq_{\text{obs}} \text{NULL} \textbf{ then} \\
\quad\quad\quad \text{secuDFS}\Big(\text{apilar}\big(\text{tope}(i) \rightarrow \text{izq}, \text{apilar}\big(\text{tope}(i) \rightarrow \text{der}, \text{desapilar}(i)\big)\big)\Big) \circ \big\langle \text{tope}(i) \rightarrow \text{clave}, \\
\quad\quad\quad *(\text{tope}(i) \rightarrow \text{significado})\big\rangle \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{if } \text{tope}(i) \rightarrow \text{der} \neq_{\text{obs}} \text{NULL} \textbf{ then} \\
\quad\quad\quad\quad \text{secuDFS}\Big(\text{apilar}\big(\text{tope}(i) \rightarrow \text{der}, \text{desapilar}(i)\big)\Big) \circ \big\langle \text{tope}(i) \rightarrow \text{clave}, *(\text{tope}(i) \rightarrow \text{significado})\big\rangle \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \textbf{if } \text{tope}(i) \rightarrow \text{izq} \neq_{\text{obs}} \text{NULL} \textbf{ then} \\
\quad\quad\quad\quad\quad \text{secuDFS}\Big(\text{apilar}\big(\text{tope}(i) \rightarrow \text{izq}, \text{desapilar}(i)\big)\Big) \circ \big\langle \text{tope}(i) \rightarrow \text{clave}, *(\text{tope}(i) \rightarrow \\
\quad\quad\quad\quad\quad \text{significado})\big\rangle > \\
\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad \text{secuDFS}\big(\text{desapilar}(i)\big) \circ \big\langle \text{tope}(i) \rightarrow \text{clave}, *(\text{tope}(i) \rightarrow \text{significado})\big\rangle > \\
\quad\quad\quad\quad \textbf{fi} \\
\quad\quad\quad \textbf{fi} \\
\quad\quad \textbf{fi} \\
\textbf{fi}
\end{array}
$$

Algoritmos

Algoritmos de diccNat

---

iVacio() → *res* : estrDiccNat

res ← *NULL* ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Apuntar a `NULL` un puntero es $O(1)$

---

iDefinir(**in** n: `nat`, **in** s: `significado`, **in/out** $dicc$: `estrDiccNat`)

```
diccAux ← dicc                                          /* Copiamos el PUNTERO */ ▷ O(1)
termine ← false                                                                   ▷ O(1)
while not termine do      ▷ O(#claves(dicc) * ...) / O(log #claves(dicc) * ...) promedio con claves uniformes
    if diccAux = NULL then                                                        ▷ O(1)
        dicc ← &⟨n, &s, NULL, NULL⟩                                               ▷ O(1)
        termine ← true                                                            ▷ O(1)
    else
        if (diccAux → clave) < n then                                             ▷ O(1)
            if not diccAux → izq = NULL then                                      ▷ O(1)
                diccAux ← (diccAux → izq)                   /* Copiamos el PUNTERO */ ▷ O(1)
            else
                (diccAux → izq) ← &⟨n, &s, NULL, NULL⟩                            ▷ O(1)
                termine ← true                                                    ▷ O(1)
            end if
        else
            if not diccAux → der = NULL then                                      ▷ O(1)
                diccAux ← (diccAux → der)                   /* Copiamos el PUNTERO */ ▷ O(1)
            else
                (diccAux → der) ← &⟨n, &s, NULL, NULL⟩                            ▷ O(1)
                termine ← true                                                    ▷ O(1)
            end if
        end if
    end if
end while
```

Complejidad:

En el peor caso: $O(\#claves(dicc))$

En promedio: $O(log\ \#claves(dicc))$

Justificación:

En el peor caso se definen todas las claves ordenadas y queda un árbol derechista o izquierdista, por eso para definir tenemos que recorrer todas las claves y tenemos $O(n)$, donde n es la cantidad de claves del diccionario.

Pero como el enunciado del trabajo práctico dice que los valores nat se insertan con probabilidad uniforme, cada vez que bajemos un nivel en la altura del árbol nos vamos a quedar con la mitad de claves, por lo tanto tenemos una complejidad de $O(log\ n)$ en promedio, donde n es la cantidad de claves del diccionario.

iDef?(**in** n: `nat`, **in** $dicc$: `estrDiccNat`) $\rightarrow$ res : bool

```
diccAux ← dicc                                          /* Copiamos el PUNTERO */ ▷ O(1)
termine ← false                                                                   ▷ O(1)
while not termine do        ▷ O(#claves(dicc) * ...) / O(log #claves(dicc) * ...) promedio con claves uniformes
    if diccAux = NULL then                                                        ▷ O(1)
        res ← false                                                               ▷ O(1)
        termine ← true                                                            ▷ O(1)
    else
        if (diccAux → clave) = n then                                             ▷ O(1)
            termine ← true                                                        ▷ O(1)
            res ← true                                                            ▷ O(1)
        else
            if (diccAux → clave) < n then                                         ▷ O(1)
                diccAux ← (diccAux → izq)                   /* Copiamos el PUNTERO */ ▷ O(1)
            else
                diccAux ← (diccAux → der)                   /* Copiamos el PUNTERO */ ▷ O(1)
            end if
        end if
    end if
end while
```

Complejidad:

En el peor caso: $O(\#claves(dicc))$

En promedio: $O(log\ \#claves(dicc))$

Justificación:

En el peor caso se definen todas las claves ordenadas y queda un árbol derechista o izquierdista, por eso para buscar si está definido, tenemos que recorrer todas las claves y tenemos $O(n)$, donde n es la cantidad de claves del diccionario.

Pero como el enunciado del trabajo práctico dice que los valores nat se insertan con probabilidad uniforme, cada vez que bajemos un nivel en la altura del árbol nos vamos a quedar con la mitad de claves, por lo tanto tenemos una complejidad de $O(log\ n)$ en promedio, donde n es la cantidad de claves del diccionario.

iObtener(**in** n: `nat`, **in** $dicc$: `estrDiccNat`) $\rightarrow$ res : significado

```
diccAux ← dicc                                   /* Copiamos el PUNTERO */ ▷ O(1)
termine ← false                                                            ▷ O(1)
while not termine do      ▷ O(#claves(dicc) * ...) / O(log #claves(dicc) * ...) promedio con claves uniformes
    if (diccAux → clave) < n then                                          ▷ O(1)
        diccAux ← (diccAux → izq)                /* Copiamos el PUNTERO */ ▷ O(1)
    else
        if (diccAux → clave) = n then                                      ▷ O(1)
            res ← (diccAux → significado)        /* lo pasamos por referencia */ ▷ O(1)
        else
            diccAux ← (diccAux → der)            /* Copiamos el PUNTERO */ ▷ O(1)
        end if
    end if
end while
```

Complejidad:

En el peor caso: $O(\#claves(dicc))$

En promedio: $O(log\ \#claves(dicc))$

Justificación:

En el peor caso se definen todas las claves ordenadas y queda un árbol derechista o izquierdista, por eso para obtener el significado, tenemos que recorrer todas las claves y tenemos $O(n)$, donde n es la cantidad de claves del diccionario.

Pero como el enunciado del trabajo práctico dice que los valores nat se insertan con probabilidad uniforme, cada vez que bajemos un nivel en la altura del árbol nos vamos a quedar con la mitad de claves, por lo tanto tenemos una complejidad de $O(log\ n)$ en promedio, donde n es la cantidad de claves del diccionario.

iBorrar(**in** n: `nat`, **in/out** $dicc$: `estrDiccNat`)

```
diccAux ← dicc                                          /* Copiamos el PUNTERO */ ▷ O(1)
termine ← false                                                                   ▷ O(1)
padre ← NULL                                                                      ▷ O(1)
while not termine do   ▷ O(#claves(dicc) * ...) / O(log #claves(dicc) * ...) promedio con claves uniformes
    if (diccAux → clave) < n then                                                 ▷ O(1)
        padre ← diccAux                                                           ▷ O(1)
        diccAux ← (diccAux → izq)                           /* Copiamos el PUNTERO */ ▷ O(1)
    else
        if (diccAux → clave) = n then                                             ▷ O(1)
            termine ← true                                                        ▷ O(1)
        else
            padre ← diccAux                                                       ▷ O(1)
            diccAux ← (diccAux → der)                       /* Copiamos el PUNTERO */ ▷ O(1)
        end if
    end if
end while

// Caso hoja
if (diccAux → izq) = NULL and (diccAux → der) = NULL then                         ▷ O(1)
    if (padre → izq) = diccAux then                                               ▷ O(1)
        (padre → izq) ← NULL                                                      ▷ O(1)
    else
        (padre → der) ← NULL                                                      ▷ O(1)
    end if
    // Caso un sólo hijo (derecho)
else if (diccAux → izq) = NULL and not (diccAux → der) = NULL then                ▷ O(1)
    if (padre → izq) = diccAux then                                               ▷ O(1)
        (padre → izq) ← (diccAux → der)                                           ▷ O(1)
    else
        (padre → der) ← (diccAux → der)                                           ▷ O(1)
    end if
    // Caso un sólo hijo (izquierdo)
else if not (diccAux → izq) = NULL and (diccAux → der) = NULL then                ▷ O(1)
    if (padre → izq) = diccAux then                                               ▷ O(1)
        (padre → izq) ← (diccAux → izq)                                           ▷ O(1)
    else
        (padre → der) ← (diccAux → izq)                                           ▷ O(1)
    end if
    // Caso dos hijos
else if not (diccAux → izq) = NULL and not (diccAux → der) = NULL then   ▷ 2 * O(log #claves(dicc))
    temp ← Min(diccAux → der)
                    ▷ O(#claves(dicc) * ...) / O(log #claves(dicc) * ...) promedio con claves uniformes
    Borrar(temp.clave, dicc) // entra en caso hoja o caso un sólo hijo por características de mínimo

    (diccAux → clave) ← temp.clave                                                ▷ O(1)
    (diccAux → significado) ← temp.significado                                    ▷ O(1)
end if
```

Complejidad:

En el peor caso: $O(\#claves(dicc))$

En promedio: $O(log\ \#claves(dicc))$

Justificación:

En el peor caso se definen todas las claves ordenadas y queda un árbol derechista o izquierdista, por eso para borrar la clave, tenemos que recorrer todas las claves y tenemos $O(n)$, donde n es la cantidad de claves del diccionario.

Pero como el enunciado del trabajo práctico dice que los valores nat se insertan con probabilidad uniforme, cada vez que bajemos un nivel en la altura del árbol nos vamos a quedar con la mitad de claves, por lo tanto tenemos una complejidad de $O(log\ n)$ en promedio, donde n es la cantidad de claves del diccionario.

iMin(**in** *dicc*: `estrDiccNat`) → *res* : tupla⟨nat, significado⟩

diccAux ← *dicc* ▷ $O(1)$
while not $(diccAux \to izq) = NULL$ **do**
▷ $O(\#claves(dicc) * ...)$ / $O(log\ \#claves(dicc) * ...)$ *promedio con claves uniformes*
diccAux ← $(diccAux \to izq)$ ▷ $O(1)$
end while
$res \leftarrow \langle diccAux \to clave, diccAux \to significado \rangle$ ▷ $O(1)$

Complejidad:
En el peor caso: $O(\#claves(dicc))$
En promedio: $O(log\ \#claves(dicc))$
Justificación:
En el peor caso se definen todas las claves ordenadas y queda un árbol derechista o izquierdista, por eso para encontrar la clave mínima, tenemos que recorrer todas las claves y tenemos $O(n)$, donde n es la cantidad de claves del diccionario.
Pero como el enunciado del trabajo práctico dice que los valores nat se insertan con probabilidad uniforme, cada vez que bajemos un nivel en la altura del árbol nos vamos a quedar con la mitad de claves, por lo tanto tenemos una complejidad de $O(log\ n)$ en promedio, donde n es la cantidad de claves del diccionario.

---

iMax(**in** *dicc*: `estrDiccNat`) → *res* : tupla⟨nat, significado⟩

diccAux ← *dicc* ▷ $O(1)$
while not $(diccAux \to der) = NULL$ **do**
▷ $O(\#claves(dicc) * ...)$ / $O(log\ \#claves(dicc) * ...)$ *promedio con claves uniformes*
diccAux ← $(diccAux \to der)$ ▷ $O(1)$
end while
$res \leftarrow \langle diccAux \to clave, diccAux \to significado \rangle$ ▷ $O(1)$

Complejidad:
En el peor caso: $O(\#claves(dicc))$
En promedio: $O(log\ \#claves(dicc))$
Justificación:
En el peor caso se definen todas las claves ordenadas y queda un árbol derechista o izquierdista, por eso para encontrar la clave máxima, tenemos que recorrer todas las claves y tenemos $O(n)$, donde n es la cantidad de claves del diccionario.
Pero como el enunciado del trabajo práctico dice que los valores nat se insertan con probabilidad uniforme, cada vez que bajemos un nivel en la altura del árbol nos vamos a quedar con la mitad de claves, por lo tanto tenemos una complejidad de $O(log\ n)$ en promedio, donde n es la cantidad de claves del diccionario.

Algoritmos del iterador

---

iCrearIt(**in** *dicc*: `estrDiccNat`) → *res* : itDiccNat

res ← $Vacia()$ ▷ $O(1)$
if not $dicc = NULL$ **then** ▷ $O(1)$
$Apilar(res, *dicc)$ ▷ $O(copy(*dicc)) = O(1)$
end if

Complejidad: $O(1)$
Justificación: Se llama únicamente a la función apilar del módulo pila del apunte de módulos básicos. Como se copia un nat la complejidad es $O(1)$.

---

iSiguientes(**in** it : `iter`) $\rightarrow$ res : lista$\big($tupla$\langle$nat, significado$\rangle\big)$

```
// DFS
res ← Vacia()                                                          ▷ O(1)
iterador ← Copiar(it)
while not EsVacia?(iterador) do                                        ▷ O(n * copy(significado mas costoso))
    prox ← Desapilar(iterador)                                         ▷ O(1)
    AgregarAtras(res, ⟨prox → clave, prox → significado⟩)              ▷ O(copy(tupla⟨nat, significado⟩))
    if not prox → der = NULL then                                      ▷ O(1)
        Apilar(iterador, prox → der)                                   ▷ O(1)
    end if
    if not prox → izq = NULL then                                      ▷ O(1)
        Apilar(iterador, prox → izq)                                   ▷ O(1)
    end if
end while
```

Complejidad: $O(n * copy(x))$ con n siendo la cantidad de claves del diccionario y x siendo el significado más costoso de copiar.
Justificación: Tiene un ciclo principal y todas las demás operaciones $O(1)$. En el ciclo principal se itera n veces y se copia en cada iteración una tupla<nat, significado>. Como copiar un nat es $O(1)$, nos queda una complejidad de $O\big(n * copy(significado\ mas\ costoso)\big)$

---

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

```
prox ← Desapilar(it)                                                   ▷ O(1)
if not prox → der = NULL then                                          ▷ O(1)
    Apilar(it, prox → der)                                             ▷ O(copy(puntero)) = O(1) (al menos una sola vez)
end if
if not prox → izq = NULL then                                          ▷ O(1)
    Apilar(it, prox → izq)                                             ▷ O(1)
end if
```

Complejidad: $O(1)$
Justificación: La funcion desapilar es $O(1)$, luego se llama dos veces a la función apilar del modulo basico pila que tiene una complejidad de $O\big(copy(\alpha)\big)$, pero como α en este algoritmo es un puntero, y copiar un puntero es $O(1)$, entonces la complejidad del algoritmo es $O(1)$.

---

iHayMas?(**in** it : `iter`) $\rightarrow$ res : bool

```
res ← (not EsVacia?(it))                                               ▷ O(1)
```

Complejidad: $O(1)$
Justificación: Se llama únicamente a la función EsVacia del módulo Pila del apunte de módulos básicos, que tiene complejidad $O(1)$.

---

iActual(**in** it : `iter`) $\rightarrow$ res : tupla$\langle$nat, significado$\rangle$

```
res ← ⟨Tope(it) → clave, Tope(it) → significado⟩)
```

Complejidad: $O(1)$
Justificación: No se copia la tupla actual del iterador, si no que se pasa por referencia. Por lo tanto es $O(1)$.

---

4. Módulo Tabla

Interfaz

se explica con: TABLA.

géneros: `tabla`.

Operaciones básicas de tabla

NOMBRE(**in** t: `tabla`) $\rightarrow$ res : `string`
Pre $\equiv$ {true}
Post $\equiv$ {alias$\big(res =_{\text{obs}} \text{nombre}(t)\big)$}
Complejidad: $O(1)$
Descripción: devuelve el nombre de la tabla indicada.
Aliasing: se pasa por referencia. No es modificable (const).

CLAVES(**in** t: `tabla`) $\rightarrow$ res : `itBi(campo)`
Pre $\equiv$ {true}
Post $\equiv$ {alias$\big(res =_{\text{obs}} \text{crearIt}(\text{claves}(t))\big)$}
Complejidad: O(1)
Descripción: devuelve un iterador al conjunto de campos claves de la tabla indicada.
Aliasing: se pasa por referencia. No es modificable (const)

BUSCAR(**in** c: `campo`, **in** d: `dato`, **in** t: `tabla`) $\rightarrow$ res : `secu(registro)`
Pre $\equiv$ {$c \in \text{campos}(t) \wedge_{\text{L}} \big(\text{tipoCampo}(c,\, t) =_{\text{obs}} \text{nat?}(d)\big)$}
Post $\equiv$ {$(\forall\ r : \text{registro})\ \text{def?}(c,\, r)\ \Rightarrow\ \big((r \in \text{registros}(t) \wedge \text{obtener}(c,\, r) =_{\text{obs}} d) \iff \text{esta?}(r,\, res)\big)$}
Complejidad:
Campo indexado nat y clave $\Rightarrow\ O\big(log\ n\ +\ |L|\big)$ promedio.
Campo indexado nat y no clave $\Rightarrow\ O\big(log\ n\ +\ n * |L|\big)$ promedio.

Campo indexado String y clave $\Rightarrow\ O(|L|\ +\ |L|) = O(|L|)$.
Campo indexado String y no clave $\Rightarrow\ O\big(|L|\ +\ n * |L|\big) = O(n * |L|)$.

Campo NO indexado $\Rightarrow\ O(n * |L|)$.

Donde n es la cantidad de registros de la tabla pasada por argumento y $|L|$ corresponde a la longitud máxima de cualquier valor string de datos de la tabla.
Descripción: Busca en todos los registros de la tabla los que tengan el dato d en el campo c, esos registros los devuelve en una secuencia.
Aliasing: no hay ya que se copian los registros.

INDICES(**in** t: `tabla`) $\rightarrow$ res : `itBi(campo)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \text{crearIt}(\text{indices}(t))$}
Complejidad: $O(1)$
Descripción: devuelve un iterador al conjunto de campos con índice de la tabla indicada.

CAMPOS(**in** t: `tabla`) $\rightarrow$ res : `itBi(campo)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \text{crearIt}(\text{campos}(t))$}
Complejidad: $O(1)$
Descripción: devuelve un iterador al conjunto de campos (devuelto por copia) de la tabla indicada.

TIPOCAMPO(**in** c: `campo`, **in** t: `tabla`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$c \in \text{campos}(t)$}
Post $\equiv$ {$res =_{\text{obs}} \text{tipoCampo}(c,\, t)$}
Complejidad: $O(1)$
Descripción: devuelve `true` si el tipo de campo es nat y `false` si el tipo de campo es string.

REGISTROS(**in** t: `tabla`) $\rightarrow$ res : `itBi(registro)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearIt(registros(t))}
Complejidad: $O(1)$
Descripción: devuelve un iterador al conjunto de registros de la tabla indicada.
Aliasing: hay aliasing, pero no es modificable.

CANTIDADDEACCESOS(**in** t: `tabla`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ cantidadDeAccesos(t)}
Complejidad: $O(1)$
Descripción: devuelve la cantidad de accesos de la tabla indicada.

NUEVATABLA(**in** *nombre*: `string`, **in** *claves*: `conj(campo)`, **in** *columnas*: `dicc(string, bool)`) $\rightarrow$ res : `tabla`
Pre $\equiv$ {*claves* $\neq_{\text{obs}} \emptyset \wedge$ *claves* $\subseteq$ claves(*columnas*)}
Post $\equiv$ {$res =_{\text{obs}}$ nuevaTabla(*nombre*, *claves*, *columnas*)}
Complejidad: $O(1)$
Descripción: genera una tabla con los valores ingresados.

AGREGARREGISTRO(**in** r: `registro`, **in/out** t: `tabla`)
Pre $\equiv$ {campos(r) $=_{\text{obs}}$ campos(t) $\wedge$ puedoInsertar?(r, t) $\wedge$ $t_0 = t$}
Post $\equiv$ {$t =_{\text{obs}}$ agregarRegistro(r, t_0)}
Complejidad:
Campo indexado $\Rightarrow$ En caso promedio $O(|L| + log\ n)$, donde n es la cantidad de registros(t) y L es el string más largo de r.
Campo no indexado $\Rightarrow$ $O(|S|)$, donde S es el string más largo de r.
Descripción: agrega un registro a la tabla.

BORRARREGISTRO(**in** *criterio*: `registro`, **in/out** t: `tabla`)
Pre $\equiv$ {#campos(*criterio*) $=_{\text{obs}}$ 1 $\wedge_{\text{L}}$ dameUno$\big($campos(*criterio*)$\big) \in$ claves(t) $\wedge$ $t_0 = t$}
Post $\equiv$ {$t =_{\text{obs}}$ borrarRegistro(*criterio*, t_0)}
Complejidad:
Criterio sobre campo indexado $\Rightarrow$ $O(log\ n\ +\ L)$.
Criterio sobre campo no indexado $\Rightarrow$ $O(n * |L|)$.
Donde n es la cantidad total de registros de la tabla y L el valor string más largo de todos los datos comparados.
Descripción: borra un registro de la tabla.

INDEXAR(**in** c: `campo`, **in/out** t: `tabla`)
Pre $\equiv$ {puedeIndexar(c, t)}
Post $\equiv$ {$t =_{\text{obs}}$ indexar(c, t)}
Complejidad: $O\big(|registros| * L * (L + log\ |registros|)\big)$, donde L es el máximo string para el campo c en cualquier registro.
Descripción: indexa un campo de la tabla.

MINIMO(**in** c: `campo`, **in** t: `tabla`) $\rightarrow$ res : `dato`
Pre $\equiv$ {$\neg$vacio?(registros(t)) $\wedge$ $c \in$ indices(t)}
Post $\equiv$ {alias($res =_{\text{obs}} m$) | nat?(m) $\Rightarrow$ valorNat(m) $=_{\text{obs}}$ valorNat$\big($minimo(c, t)$\big)$ $\wedge$ $\neg$nat?(m) $\Rightarrow$ valorStr(m) $=_{\text{obs}}$ valorStr$\big($minimo(c, t)$\big)$}
Complejidad: $O(1)$
Descripción: devuelve el minimo de una tabla por referencia de un campo indexado. res no es modificable
Aliasing: res no es modificable.

MAXIMO(**in** c: `campo`, **in** t: `tabla`) $\rightarrow$ res : `dato`
Pre $\equiv$ {$\neg$vacio?(registros(t)) $\wedge$ $c \in$ indices(t)}
Post $\equiv$ {alias($res =_{\text{obs}} m$) | tipoCampo(c, m) $\Rightarrow$ $\Big($nat?(m) $\wedge$ $\big($valorNat(m) $=_{\text{obs}}$ valorNat(maximo$\big($c, $t$$\big)$$\big)$$\Big)$ $\wedge$ $\neg$tipoCampo(c, m) $\Rightarrow$ $\Big(\neg$nat?(m) $\wedge$ $\big($valorStr(m) $=_{\text{obs}}$ valorStr(maximo$\big($c, $t$$\big)$$\big)$$\Big)$}
Complejidad: $O(1)$
Descripción: devuelve el maximo de una tabla por referencia de un campo indexado. res no es modificable

Aliasing: *res* no es modificable.

Representación

Representación de tabla

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

donde `estrTabla` es `tupla` ⟨ *indicesString*: `diccString(conj(itConj(registro)))`, *indicesNat*: `diccNat(conj(itConj(registro)))`, *registros*: `conj(registro)`, *nombre*: `string`, *campos*: `diccString(bool esNat?)`, *claves*: `conj(campo)`, *campoIndexadoNat*: `lista(tupla<`*nombre*: `campo`, *max*: `dato`, *min*: `dato`, *vacio?*: `bool>)`, *campoIndexadoString*: `lista(tupla<`*nombre*: `campo`, *max*: `dato`, *min*: `dato`, *vacio?*: `bool>)`, *cantAccesos*: `nat` ⟩

donde `registro` es `diccString(dato)` y se explica con Registro, y `conj` corresponde al conjunto lineal de la cátedra.

Invariante de representación

1) Las claves de indicesString corresponden al valor del campo indexado para cada registro que esté en sus significados.
2) Las claves de indicesNat corresponden al valor del campo indexado para cada registro que esté en sus significados.
3) Los significados de indicesString e indicesNat pertenecen a registros.
4) Todos los registros estan indexados.
5) Claves esta entre los campos y no es vacio.
6) Todos los valores de los registros son menores o iguales al campo maximo y mayor o iguales al minimo.
7) Para cada campo indexado, hay un registro cuyo valor en ese campo es el maximo y un registro cuyo valor es el minimo.
8) Si un campo es clave no puede haber dos registros con mismo dato en ese campo.
9) El tipo de dato en registro corresponde al tipo de dato en campos y las claves de los registros son los campos de la tabla.
10) El campo indexado pertenece a campos.
11) cantAccesos es menor o igual a la cantidad de registros.
12) Tamaño de las listas 'campoIndexado...' es menor o igual a 1.
13) El bool 'vacio?' de las tuplas de campoIndexado valen true si y solo si sus respectivos diccionarios están vacíos.
14) Si no hay un campoIndexado de cierto tipo, el diccionario correspodiente, esta vacío.

$$\text{Rep} : \text{estrTabla} \longrightarrow \text{bool}$$

$$\text{Rep}(e) \equiv \text{true} \iff$$

1) $\big(\neg\text{vacia?}(e.\text{campoIndexadoString})\big) \Rightarrow$
$\Big(\big(\forall\, c : \text{string},\, c \in \text{claves}(e.\text{indicesString})\big)\ \big(\forall\, r : \text{itConj(registro)},\, r \in \text{obtener}(c,\, e.\text{indicesString})\big)$
$\big(\text{valorStr}\big(\text{obtener}\big(\text{campoIndexString},\, \text{siguiente}(r)\big)\big) = c\big)\Big)$

2) $\big(\neg\text{vacia?}(e.\text{campoIndexadoNat})\big) \Rightarrow$
$\Big(\big(\forall\, c : \text{nat},\, c \in \text{claves}(e.\text{indicesNat})\big)\ \big(\forall\, r : \text{itConj(registro)},\, r \in \text{obtener}(c,\, e.\text{indicesNat})\big)$
$\big(\text{valorNat}\big(\text{obtener}\big(\text{campoIndexNat},\, \text{siguiente}(r)\big)\big) = c\big)\Big)$

3) $\big(\neg\text{vacia?}(e.\text{campoIndexadoString})\big) \Rightarrow$
$\Big(\big(\forall\, c : \text{string},\, c \in \text{claves}(e.\text{indicesString})\big)\ \big(\forall\, r : \text{itConj(registro)},\, r \in \text{obtener}(c,\, e.\text{indicesString})\big)$
$\big(\text{siguiente}(r) \in e.\text{registros}\big)\Big)$

3 bis) $\big(\neg\text{vacia?}(e.\text{campoIndexadoNat})\big) \Rightarrow$
$\Big(\big(\forall\, c : \text{nat},\, c \in \text{claves}(e.\text{indicesNat})\big)\ \big(\forall\, r : \text{itConj(registro)},\, r \in \text{obtener}(c,\, e.\text{indicesNat})\big)$
$\big(\text{siguiente}(r) \in e.\text{registros}\big)\Big)$

4) $\big(\neg\text{vacia?}(e.\text{campoIndexadoString})\big) \Rightarrow$
$\Big(\big(\forall\, r : \text{registro},\, r \in e.\text{registros}\big)\ \big(\exists\, it : \text{itConj(registro)},\, \text{siguiente}(it) =_{\text{obs}} r\big)$
$it \in \text{obtener}\big(\text{valorStr}\big(\text{obtener}(\text{campoIndexString},\, r)\big),\, e.\text{indicesString}\big)\Big)$

4 bis) $\big(\neg\text{vacia?}(e.\text{campoIndexadoNat})\big) \Rightarrow$
$\Big(\big(\forall\, r : \text{registro},\, r \in e.\text{registros}\big)\ \big(\exists\, it : \text{itConj(registro)},\, \text{siguiente}(it) =_{\text{obs}} r\big)$
$it \in \text{obtener}\big(\text{valorNat}\big(\text{obtener}(\text{campoIndexNat},\, r)\big),\, e.\text{indicesNat}\big)\Big)$

5) $\big(\forall\ c : \text{campo},\ c \in e.\text{claves}\big)\ \big(c \in \text{claves}(e.\text{campos}) \wedge \#e.\text{claves} > 0\big)$
6) $\Big(\big(\neg\text{vacia?}(e.\text{campoIndexadoNat})\big) \wedge_{\text{L}} \neg\big(\big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{vacio?}\big)\Big) \Rightarrow$
$\left(\begin{array}{l}\big(\forall\ r : \text{registro},\ r \in e.\text{registros}\big)\ \big(\forall\ c : \text{campo},\ c \in \text{claves}(e.\text{campos}) \wedge_{\text{L}} \text{obtener}(c,\ e.\text{campos}) =_{\text{obs}} \text{true}\ \big) \\ \Big(\big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{min} \leq \text{obtener}(c,\ r)\ \big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{max}\Big)\end{array}\right)$
6 bis) $\Big(\big(\neg\text{vacia?}(e.\text{campoIndexadoString})\big) \wedge_{\text{L}} \neg\big(\big(\text{prim}(e.\text{campoIndexadoString})\big).\text{vacio?}\big)\Big) \Rightarrow$
$\left(\begin{array}{l}\big(\forall\ r : \text{registro},\ r \in e.\text{registros}\big)\ \big(\forall\ c : \text{campo},\ c \in \text{claves}(e.\text{campos}) \wedge_{\text{L}} \text{obtener}(c,\ e.\text{campos}) =_{\text{obs}} \text{false}\ \big) \\ \Big(\big(\text{prim}(e.\text{campoIndexadoString})\big).\text{min} \leq \text{obtener}(c,\ r)\ \big(\text{prim}(e.\text{campoIndexadoString})\big).\text{max}\Big)\end{array}\right)$
7) $\Big(\big(\neg\text{vacia?}(e.\text{campoIndexadoString})\big) \wedge_{\text{L}} \neg\big(\big(\text{prim}(e.\text{campoIndexadoString})\big).\text{vacio?}\big)\Big) \Rightarrow$
$\left(\begin{array}{l}\big(\exists\ r,\ r' : \text{registro},\ r \in e.\text{registros} \wedge r' \in e.\text{registros}\big) \\ \Big(\text{obtener}(\text{campoIndexString},\ r) =_{\text{obs}} \big(\text{prim}(e.\text{campoIndexadoString})\big).\text{max} \wedge \\ \text{obtener}(\text{campoIndexString},\ r') =_{\text{obs}} \big(\text{prim}(e.\text{campoIndexadoString})\big).\text{min}\Big)\end{array}\right)$
7 bis) $\Big(\big(\neg\text{vacia?}(e.\text{campoIndexadoNat})\big) \wedge_{\text{L}} \neg\big(\big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{vacio?}\big)\Big) \Rightarrow$
$\left(\begin{array}{l}\big(\exists\ r,\ r' : \text{registro},\ r \in e.\text{registros} \wedge r' \in e.\text{registros}\big) \\ \Big(\text{obtener}(\text{campoIndexNat},\ r) =_{\text{obs}} \big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{max} \wedge \\ \text{obtener}(\text{campoIndexNat},\ r') =_{\text{obs}} \big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{min}\Big)\end{array}\right)$
8) $\big(\forall\ c : \text{campo},\ c \in e.\text{claves}\big)\ \big(\forall\ x,\ y : \text{registro},\ x \in e.\text{registros} \wedge y \in e.\text{registros} \wedge (x \neq_{\text{obs}} y)\big)$
$\text{obtener}(c,\ y) \neq_{\text{obs}} \text{obtener}(c,\ x)$
9) $\big(\forall\ r : \text{registro},\ r \in e.\text{registros}\big)\ \big(\forall\ c : \text{campo}\big)\ \big(c \in \text{claves}(e.\text{campos})\big) \iff$
$\big(c \in \text{claves}(r) \wedge_{\text{L}} \text{obtener}(c,\ e.\text{campos}) =_{\text{obs}} \text{tipo?}(\text{obtener}(c,r))\big)$
10) $\neg\big(\text{vacia?}(e.\text{campoIndexadoString})\big) \Rightarrow \big(\text{prim}(e.\text{campoIndexadoString})\big).\text{nombre} \in \text{claves}(e.\text{campos})$
10 bis) $\neg\big(\text{vacia?}(e.\text{campoIndexadoNat})\big) \Rightarrow \big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{nombre} \in \text{claves}(e.\text{campos})$
11) $e.\text{cantAccesos} \geq \#e.\text{registros}$
12) $\text{long}(e.\text{campoIndexadoNat}) \leq 1 \wedge \text{long}(e.\text{campoIndexadoNat}) \leq 1$
13) $\neg\big(\text{vacia?}(e.\text{campoIndexadoNat})\big) \Rightarrow$
$\big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{vacio?} \iff \big(\#\text{claves}(e.\text{indicesNat}) =_{\text{obs}} 0\big) \wedge$
$\neg\big(\text{vacia?}(e.\text{campoIndexadoString})\big) \Rightarrow$
$\big(\text{prim}(e.\text{campoIndexadoString})\big).\text{vacio?} \iff \big(\#\text{claves}(e.\text{indicesString}) =_{\text{obs}} 0\big)$
14) $e.\text{campoIndexadoNat} =_{\text{obs}} <> \Rightarrow\ \text{claves}(e.\text{indicesNat}) =_{\text{obs}} \emptyset \wedge$
$e.\text{campoIndexadoString} =_{\text{obs}} <> \Rightarrow\ \text{claves}(e.\text{indicesString}) =_{\text{obs}} \emptyset$

Auxiliares sintácticos

$\text{campoIndexString} = \Pi_1\big(\text{prim}(e.\text{campoIndexString})\big)$
$\text{campoIndexNat} = \Pi_1\big(\text{prim}(e.\text{campoIndexadoNat})\big)$

$\text{Abs} : \text{estrTabla}\ e \longrightarrow \text{tabla}$ {Rep(e)}
$\text{Abs}(e) =_{\text{obs}} \text{t: tabla} \mid \text{nombre}(t) =_{\text{obs}} e.\text{nombre} \wedge \text{claves}(t) =_{\text{obs}} e.\text{claves} \wedge \text{campos}(t) =_{\text{obs}} \text{claves}(e.\text{campos})$
$\wedge_{\text{L}}\ \big(\forall\ c : \text{campo},\ c \in \text{campos}(t)\big)\ \text{tipoCampo}(c,\ t) =_{\text{obs}} \text{obtener}(c,\ e.\text{campos}) \wedge \text{registros}(t)$
$=_{\text{obs}} e.\text{registros} \wedge \text{cantidadDeAccesos}(t) =_{\text{obs}} e.\text{cantAccesos} \wedge \big(\forall\ i : \text{campo},\ i \in \text{indices}(t)\big)$
$\Big(\text{tipoCampo}(i,\ t)\ \Rightarrow\ \big(i =_{\text{obs}} \big(\text{prim}(e.\text{campoIndexadoNat})\big).\text{nombre}\big) \wedge$
$\neg\big(\text{tipoCampo}(i,\ t)\big)\ \Rightarrow\ \big(i =_{\text{obs}} \big(\text{prim}(e.\text{campoIndexadoString})\big).\text{nombre}\big)\Big)$

Algoritmos

Algoritmos de tabla

iNombre(**in** e: `estrTabla`) $\rightarrow$ res : string

$res \leftarrow e.nombre$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: El algoritmo tiene una única llamada a una función con costo $O(1)$.

iClaves(**in** e: `estrTabla`) $\rightarrow$ res : itConj(campo)

$res \leftarrow CrearIt(e.claves)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: El algoritmo tiene una única llamada a una función con costo $O(1)$.

iIndices(**in** e: `estrTabla`) $\rightarrow$ res : itConj(campo)

$aux \leftarrow Vacio()$ ▷ $O(1)$
if $Longitud(e.campoIndexadoNat) > 0$ **then** ▷ $O(1)$
$AgregarRapido\big(Primero((e.campoIndexadoNat).nombre), aux\big)$ ▷ $O\big(copy((e.campoIndexadoNat).nombre)\big)$
end if
if $Longitud(e.campoIndexadoString) > 0$ **then** ▷ $O(1)$
$AgregarRapido\big(Primero((e.campoIndexadoString).nombre), aux\big)$
▷ $O\big(copy((e.campoIndexadoString).nombre)\big)$
end if
$res \leftarrow CrearIt(aux)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: El algoritmo utiliza la función agregarRapido del Módulo Conjunto lineal 2 veces, entonces la complejidad es la de copiar el string del campo más largo de los dos. Como los strings de los campos estan acotados por una constante, entonces la complejidad queda $O(1)$.

iCampos(**in** e: `estrTabla`) $\rightarrow$ res : itConj(campo)

$res \leftarrow CrearIt(e.campos)$ ▷ $O\big(\#claves(e.campos) * L\big)$

Complejidad: $O\big(\#claves(e.campos)\big)$
Justificación: Por módulo diccString, la operación Claves exporta complejidad $O\big(\#claves(e.campos) * L\big)$ siendo L la longitud del mayor string en claves. Dado que los nombres de los campos están acotados, la complejidad final es $O\big(\#claves(e.campos)\big)$.

iTipoCampo(**in** c: `campo`, **in** e: `estrTabla`) $\rightarrow$ res : bool

$res \leftarrow Obtener(c, e.campos)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Como la longitud de los campos es acotada, buscar en un diccString pasa de ser orden de longitud de la clave más larga a $O(1)$.

iRegistros(**in** e: `estrTabla`) $\rightarrow$ res : itConjunto(campo)

$res \leftarrow CrearIt(e.registros)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Crear un iterador del módulo conjunto lineal de la cátedra es $O(1)$.

iCantidadDeAccesos(**in** e: `estrTabla`) $\rightarrow$ res : nat

$res \leftarrow e.cantAccesos$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: El algoritmo tiene una única llamada a una función con costo $O(1)$.

iBorrarRegistro(**in** $criterio$: `registro`, **in/out** e: `estrTabla`)

$it \leftarrow CrearIt(VistaDicc(criterio))$ ▷ $O(1)$
$clave \leftarrow Siguiente(it).clave$ ▷ $O(1)$
$dato \leftarrow Siguiente(it).significado$ ▷ $O(1)$

// Si el criterio es un índice tenemos que recorrer el conjunto de iteradores a registros con un iterador borrando todos, si no hay que recorrer registros linealmente
if $(Primero(e.campoIndexadoNat)).nombre = clave$ **then** ▷ $O(|L|)$
  if $Def?(ValorNat(dato), e.indicesNat)$ **then** ▷ $O(log\ n)$ *promedio*
    $iterador \leftarrow CrearIt(Obtener(ValorNat(dato), e.indicesNat))$ ▷ $O(log\ n)$ *promedio*
    // es clave, por lo tanto es el único en el conjunto, lo borro de e.registros:
    $EliminarSiguiente(Siguiente(iterador))$ ▷ $O(1)$
    $e.cantAccesos++$ ▷ $O(1)$
    $Borrar(ValorNat(dato), e.indicesNat)$ ▷ $O(log\ n)$ *promedio*
    $temp \leftarrow CrearIt(e.indicesNat)$ ▷ $O(1)$
    if not $HaySiguiente(temp)$ **then** ▷ $O(1)$
      $Primero(e.campoIndexadoNat).vacio? \leftarrow true$ ▷ $O(1)$
    else
      // comparamos por valorNat porque es O(1) vs comparar por dato
      if $ValorNat(dato) = ValorNat(Primero(e.campoIndexadoNat).max)$ **then** ▷ $O(1)$
        $Primero(e.campoIndexadoNat).max \leftarrow DatoNat(\Pi_1(Max(e.indicesNat)))$ ▷ $O(1)$
      end if
      if $ValorNat(dato) = ValorNat(Primero(e.campoIndexadoNat).min)$ **then** ▷ $O(1)$
        $Primero(e.campoIndexadoNat).min \leftarrow DatoNat(\Pi_1(Min(e.indicesNat)))$ ▷ $O(1)$
      end if
    end if
  end if
else if $(Primero(e.campoIndexadoString)).nombre = clave$ **then** ▷ $O(|L|)$
  if $Def?(ValorString(dato), e.indicesString)$ **then** ▷ $O(|L|)$
    $iteradorCrearIt(Obtener(ValorStr(dato), e.indicesString))$ ▷ $O(|L|)$
    $EliminarSiguiente(Siguiente(iterador))$ ▷ $O(1)$
    $e.cantAccesos++$ ▷ $O(1)$
    $Borrar(ValorStr(dato), e.indicesString)$ ▷ $O(|L|)$
    $temp \leftarrow CrearIt(e.indicesString)$ ▷ $O(1)$
    if not $HaySiguiente(temp)$ **then** ▷ $O(1)$
      $Primero(e.campoIndexadoString).vacio? \leftarrow true$ ▷ $O(1)$
    else
      if $dato = Primero(e.campoIndexadoString).max$ **then** ▷ $O(L)$
        $Primero(e.campoIndexadoString).max \leftarrow DatoString(\Pi_1(Max(e.indicesString)))$
        ▷ $O(L)$ *por ref*
      end if
      if $dato = Primero(e.campoIndexadoString).min$ **then** ▷ $O(L)$
        $Primero(e.campoIndexadoString).min \leftarrow DatoString(\Pi_1(Min(e.indicesString)))$ ▷ $O(L)$
      end if
    end if
  end if
else

```
    iter ← CrearIt(e.registros)                                          ▷ O(1)
    while HaySiguiente(iter) do                                          ▷ O(n * |L|)
        if Obtener(clave, Siguiente(iter)) = dato then                   ▷ O(|L|)
            EliminarSiguiente(iter)                                      ▷ O(1)
        end if
        Avanzar(iter)                                                    ▷ O(1)
    end while
end if
```

Complejidad:

Criterio sobre campo indexado $\Rightarrow$ $O(logn + L)$.

Criterio sobre campo no indexado $\Rightarrow$ $O(n * |L|)$.

Siendo n la cantidad total de registros de la tabla y L el valor string más largo de todos los datos comparados.

Justificación:

En peor caso sobre campo indexado recorre el diccNat o el diccString para encontrar el iterador (el conjunto significado tiene longitud 1 por ser un campo clave) al conjunto y eliminarlo. Buscar en diccNat en promedio es $O(log\ n)$ dado que se inserta con probabilidad uniforme. Buscar en diccString es $O(L)$ en el peor caso.

Se agrega el costo de actualizar el máximo, que es $O(log\ n)$ para índice nat y $O(L\ +\ L) = O(L)$ (por comparar con máximo y mínimo actual y luego buscar máximo y mínimo respectivamente) para índice string, la inserción es por referencia aún así. Por lo tanto el máximo y mínimo se actualiza en $O(log\ n\ +\ L)$.

En el peor caso sobre campo no indexado debe recorrer todo el conjunto de registros de la tabla preguntando si el dato de cada registro coincide con el criterio para eliminarlo.

---

iNuevaTabla(**in** *nombre*: `string`, **in** *claves*: `conj(campo)`, **in** *columnas*: `diccString(bool)`) → *res* : estrTabla

```
res.indicesString ← Vacio()                                              ▷ O(1)
res.indicesNat ← Vacio()                                                 ▷ O(1)
res.registros ← Vacio()                                                  ▷ O(1)
res.nombre ← Copiar(nombre)                                              ▷ O(|nombre|)
res.campos ← Copiar(columnas)                                            ▷ O(#claves(columnas) * M)
res.claves ← Copiar(claves)                                              ▷ O(#claves * L)
res.campoIndexadoNat ← Vacio()                                           ▷ O(1)
res.campoIndexadoString ← Vacio()                                        ▷ O(1)
res.cantAccesos ← 0                                                      ▷ O(1)
```

Complejidad: $O(1)$

Justificación:

Todas las asignaciones que no usen copias son $O(1)$. Copiar el nombre es cte. porque, por enunciado los nombres de las tablas son acotados.

Copiar claves tiene complejidad $O(\#claves * L)$, donde L es el nombre más largo de cualquier clave, que se reduce a $O(1)$ porque por enunciado los nombres de los campos también son acotados y también la cantidad de campos por tabla (es decir $\#claves < n$, para algún n natural). Vale lo mismo para copiar columnas, que pasa de ser $O(\#claves(columnas) * M)$ siendo M el nombre de la clave más larga del diccionario (los significados de tipo bool se copian en $O(1)$) a ser $O(1)$ por los factores ya mencionados.

iAgregarRegistro(**in** r: `registro`, **in**/**out** e: `estrTabla`)

```
// Aumento la cantidad de accesos
e.cantAccesos++                                                          ▷ O(1)
// Agrego el registro al conjunto e.registros
it ← AgregarRapido(r, e.registros)                                       ▷ O(copy(r))
// Se paga una cantidad de veces acotada por copiar datos acotados en costo por L, que es el dato string más largo

// Me fijo si tengo un campo nat indexado (me fijo en e.campoIndexadoNat)
if not Vacia?(e.campoIndexadoNat) then                                   ▷ O(1)
    if (Primero(e.campoIndexadoNat)).vacio? then                         ▷ O(1)
        (Primero(e.campoIndexadoNat)).min ← Obtener((Primero(e.campoIndexadoNat)).nombre, r)   ▷ O(1)
        (Primero(e.campoIndexadoNat)).max ← (Primero(e.campoIndexadoNat)).min                  ▷ O(1)
        (Primero(e.campoIndexadoNat)).vacio? ← false                     ▷ O(1)
    else
        nPaMinMax ← Obtener((Primero(e.campoIndexadoNat)).nombre, r)     ▷ O(1)
        if nPaMinMax < (Primero(e.campoIndexadoNat)).min then
            (Primero(e.campoIndexadoNat)).min ← nPaMinMax                ▷ O(1)
        end if
        if nPaMinMax > (Primero(e.campoIndexadoNat)).max then
            (Primero(e.campoIndexadoNat)).max ← nPaMinMax                ▷ O(1)
        end if
    end if
    aux ← Obtener((Primero(e.campoIndexadoNat)).nombre, r)               ▷ O(1)
    // Si lo tengo indexado y está definido
    if Def?(aux, e.indicesNat) then                                      ▷ O(log n)
        AgregarRapido(it, Obtener(aux, e.indicesNat))                    ▷ O(copy(it))
    else
        Definir(aux, AgregarRapido(it, Vacio()), e.indicesNat)           ▷ O(log n)
    end if
end if

if not Vacia?(e.campoIndexadoString) then                                ▷ O(1)
    if (Primero(e.campoIndexadoString)).vacio? then                      ▷ O(1)
        (Primero(e.campoIndexadoString)).min ← Obtener((Primero(e.campoIndexadoString)).nombre, r)
                                                                         ▷ O(1)
        (Primero(e.campoIndexadoString)).max ← (Primero(e.campoIndexadoString)).min   ▷ O(1)
        (Primero(e.campoIndexadoNat)).vacio? ← false                     ▷ O(1)
    else
        sPaMinMax ← Obtener((Primero(e.campoIndexadoString)).nombre, r)  ▷ O(1)
        if sPaMinMax < (Primero(e.campoIndexadoString)).min then         ▷ O(L)
            (Primero(e.campoIndexadoString)).min ← sPaMinMax             ▷ O(1)
        end if
        if sPaMinMax > (Primero(e.campoIndexadoString)).max then         ▷ O(L)
            (Primero(e.campoIndexadoString)).max ← sPaMinMax             ▷ O(1)
        end if
    end if
    aux ← Obtener((Primero(e.campoIndexadoString)).nombre, r)            ▷ O(1)
    if Def?(aux, e.indicesString) then                                   ▷ O(Max string)
        AgregarRapido(it, Obtener(aux, e.indicesString))                 ▷ O(copy(it))
    else
        Definir(aux, AgregarRapido(it, Vacio()), e.indicesString)        ▷ O(Max string)
    end if
end if
```

Complejidad:

Campo indexado: $O(|L| + log\ n)$

Campo no indexado: $O(|S|)$

Donde n es la cantidad de claves del diccNat e.indicesNat (acotada por la cantidad de registros de la tabla) y L es el string más largo de cualquier registro en la tabla. S es el string más largo del registro a agregar.

Justificación:

Campo indexado: Agregar el registro al conjunto cuesta $O(copy(r))$; al ser un diccString eso sería $O(\#claves(r) * Max\{K, S\})$, siendo K la clave de máximo costo para copiar y S lo mismo pero para significados. Como la cantidad de claves está acotada por haber una cantidad acotada de campos (por enunciado) y la longitud de los nombres de campos también, vale que $O(\#claves(r) * K) = O(1)$. Por lo tanto, el peor caso es pagar por el copiado del significado más costoso, que corresponde a la longitud máxima de cualquier string del registro (que acotamos por la máxima longitud de cualquier string de cualquier registro de la tabla, y lo denominamos $|L|$).

Además se agrega el costo de agregar en el diccionario de índices nat (logarítmico en la cantidad n de registros de la tabla) y el de agregar en el diccionario de índices string (nuevamente, longitud máxima de cualquier string de cualquier registro de la tabla, es decir $O(|L|)$).

Por lo tanto, la complejidad final queda $O(|L| + log\ n + |L|) = O(|L| + log\ n)$.

Campo no indexado: Agregar el registro al conjunto cuesta $O(copy(r))$, esto es igual a copiar el string más largo ya que copiar los nat es $O(1)$. Luego el algoritmo si no hay campos indexados no hace mas operaciones que sean mayores a $O(1)$. Por lo tanto el algoritmo tiene complejidad $O(|S|)$, siendo S el string más largo del registro a agregar.

---

iIndexar(**in** c: `campo`, **in/out** e: `estrTabla`)

```
// si es tipo nat...
if TipoCampo(c, e) then                                                          ▷ O(1)
    dato ← DatoNat(0)                                                            ▷ O(1)
    // Agrego adelante de la lista de campoIndexadoNat
    AgregarAdelante(e.campoIndexadoNat, ⟨c, dato, dato, true⟩)   ▷ O(copy(⟨c, dato, dato, bool⟩)) ∈ O(1)

    it ← CrearIt(e.registros)                                                    ▷ O(1)
    // Si hay algun registro entonces lo seteo como maximo y minimo y en el while pregunto
    if HaySiguiente(it) then                                                     ▷ O(1)
        (Primero(e.campoIndexadoNat)).vacio? ← false                             ▷ O(1)
        (Primero(e.campoIndexadoNat)).max? ← Obtener(c, Siguiente(it))           ▷ O(1)
        (Primero(e.campoIndexadoNat)).min? ← Obtener(c, Siguiente(it))           ▷ O(1)
    end if
    while HaySiguiente(it) do                                                    ▷ O(#registros(e) * ...)
        temp ← ValorNat(Obtener(c, Siguiente(it)))                               ▷ O(1)
        if not Def?(Obtener(temp, e.indicesNat)) then                            ▷ O(log #registros)
            Definir(temp, Vacio(), e.indicesNat)                                 ▷ O(log #registros)
        end if
        AgregarRapido(it, Obtener(temp, e.indicesNat))          ▷ O(copy(it) + log #registros)
        if Obtener(c, Siguiente(it)) > (Primero(e.campoIndexadoNat)).max then    ▷ O(1)
            (Primero(e.campoIndexadoNat)).max ← Obtener(c, Siguiente(it))        ▷ O(1)
        end if
        if Obtener(c, Siguiente(it)) < (Primero(e.campoIndexadoNat)).min then    ▷ O(L)
            (Primero(e.campoIndexadoNat)).min ← Obtener(c, Siguiente(it))        ▷ O(1)
        end if
        Avanzar(it)                                                              ▷ O(1)
    end while
else
```

```
dato ← DatoStr("temp")                                                          ▷ O(1)
AgregarAdelante(e.campoIndexadoString, ⟨c, dato, dato, true⟩)                   ▷ O(copy(⟨c, dato, dato, bool⟩)) ∈ O(1)

it ← CrearIt(e.registros)                                                       ▷ O(1)
// Si hay algun registro entonces lo seteo como maximo y minimo y en el while pregunto
if HaySiguiente(it) then                                                        ▷ O(1)
    (Primero(e.campoIndexadoString)).vacio? ← false                             ▷ O(1)
    (Primero(e.campoIndexadoString)).max? ← Obtener(c, Siguiente(it))           ▷ O(1)
    (Primero(e.campoIndexadoString)).min? ← Obtener(c, Siguiente(it))           ▷ O(1)
end if
while HaySiguiente(it) do                                                       ▷ O(#registros(e) ∗ ...)
    temp ← ValorStr(Obtener(c, Siguiente(it)))                                  ▷ O(1)
    if not Def?(Obtener(temp, e.indicesString)) then                            ▷ O(|L|)
        Definir(temp, Vacio(), e.indicesString)                                 ▷ O(|L|)
    end if
    AgregarRapido(it, Obtener(temp, e.indicesString))                           ▷ O(copy(it) + log #registros)
    if Obtener(c, Siguiente(it)) > (Primero(e.campoIndexadoString)).max then    ▷ O(1)
        (Primero(e.campoIndexadoString)).max ← Obtener(c, Siguiente(it))        ▷ O(1)
    end if
    if Obtener(c, Siguiente(it)) < (Primero(e.campoIndexadoString)).min then    ▷ O(L)
        (Primero(e.campoIndexadoString)).min ← Obtener(c, Siguiente(it))        ▷ O(1)
    end if
    Avanzar(it)                                                                 ▷ O(1)
end while
end if
```

Complejidad: $O(|registros| * L * (L + log\ |registros(e)|))$, donde L es el máximo string para el campo c en cualquier registro.

Justificación:

En el peor caso se recorren todos los registros definiendo un iterador suyo $(O(1))$ en un diccString (inserción en $O(L)$) o insertando en un diccNat (en $O(log\ |registros|)$ para caso promedio), por el costo de copiar cada valorStr si es máximo o mínimo (acotado por L).

iBuscar(**in** c: `campo`, **in** d: `dato`, **in** e: `estrTabla`) $\rightarrow$ res : lista(registro)

```
res ← Vacia()                                                                  ▷ O(1)
if thenNat?(d)                                                                 ▷ O(1)
    // caso campoJOIN, donde esta indexado
    if (Primero(e.campoIndexadoNat)).nombre = c then        /* nombres acotados */ ▷ O(|c|) = O(1)
        if Def?(ValorNat(d), e.indicesNat) then             /* n cantidad de registros */ ▷ O(log n)
            itConjIts ← CrearIt(Obtener(ValorNat(d), e.indicesNat))            ▷ O(log n)
            while HaySiguiente?(itConjIts) do   ▷ O(1 * ...) si c es clave /O(n * ...) si no
                AgregarAtras(Siguiente(Siguiente(itConjIts)), res)   /* L mayor string de la tabla */ ▷
O(#campos * |L|) = O(|L|)
                Avanzar(itConjIts)                                             ▷ O(1)
            end while
        end if
    else
        it ← CrearIt(e.registros)                                              ▷ O(1)
        while HaySiguiente?(it) do                                             ▷ O(n * ...)
            if Obtener(c, Siguiente(it)) = d then                              ▷ O(|ValorStr(d)|) = O(|L|)
                AgregarAtras(Siguiente(it), res)
            end if
            Avanzar(it)                                                        ▷ O(1)
        end while
    end if
else
    // caso campoJOIN, donde esta indexado
    if (Primero(e.campoIndexadoString)).nombre = c then     /* nombres acotados */ ▷ O(|c|) = O(1)
        if Def?(ValorStr(d), e.indicesString) then                             ▷ O(|L|)
            itConjIts ← CrearIt(Obtener(ValorStr(d), e.indicesString))         ▷ O(L)
            while HaySiguiente?(itConjIts) do   ▷ O(1 * ...) si c es clave /O(n * ...) si no
                AgregarAtras(Siguiente(Siguiente(itConjIts)), res)             ▷ O(|L|)
                Avanzar(itConjIts)                                             ▷ O(1)
            end while
        end if
    else
        it ← CrearIt(e.registros)                                              ▷ O(1)
        while HaySiguiente?(it) do                                             ▷ O(n * ...)
            if Obtener(c, Siguiente(it)) = d then                              ▷ O(|ValorStr(d)|) = O(|L|)
                AgregarAtras(Siguiente(it), res)
            end if
            Avanzar(it)                                                        ▷ O(1)
        end while
    end if
end if
```

Complejidad:

Campo indexado nat y clave $\Rightarrow$ $O(log\ n\ +\ |L|)$ promedio.
Campo indexado nat y no clave $\Rightarrow$ $O(log\ n\ +\ n * |L|)$ promedio.

Campo indexado String y clave $\Rightarrow$ $O(|L|\ +\ |L|) = O(|L|)$.
Campo indexado String y no clave $\Rightarrow$ $O(|L| + n * |L|) = O(n * |L|)$.

Campo NO indexado $\Rightarrow$ $O(n * |L|)$.
Donde n es la cantidad de registros y L el string más largo de la tabla.

Justificación: En el peor caso se recorren todos los registros, con cada caso detallado anteriormente.

---

iMinimo(**in** c: `campo`, **in** e: `estrTabla`) $\rightarrow$ res : dato

if $c = \big(Primero(e.campoIndexadoNat)\big).nombre$ **then** ▷ $O(1)$
  $res \leftarrow \big(Primero(e.campoIndexadoNat)\big).min$ ▷ $O(1)$
else
  $res \leftarrow \big(Primero(e.campoIndexadoStr)\big).min$ ▷ $O(1)$
end if

Complejidad: $O(1)$
Justificación: El resultado se devuelve por referencia.

---

iMaximo(**in** c: `campo`, **in** e: `estrTabla`) $\rightarrow$ res : dato

if $c = \big(Primero(e.campoIndexadoNat)\big).nombre$ **then** ▷ $O(1)$
  $res \leftarrow \big(Primero(e.campoIndexadoNat)\big).max$ ▷ $O(1)$
else
  $res \leftarrow \big(Primero(e.campoIndexadoStr)\big).max$ ▷ $O(1)$
end if

Complejidad: $O(1)$
Justificación: El resultado se devuelve por referencia.

---

5. Módulo Base de Datos

Interfaz

se explica con: Base de Datos.

géneros: `base`.

Operaciones básicas de base

NuevaBDD() $\rightarrow$ *res* : `base`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ nuevaBDD}
Complejidad: $O(1)$
Descripción: crea una base de datos sin tablas.

AgregarTabla(**in** t: `tabla`, **in/out** b: `base`)
Pre $\equiv$ {vacio?(registros(t)) $\wedge$ $b = b_0$}
Post $\equiv$ {$b =_{\text{obs}}$ agregarTabla(t, b_0)}
Complejidad: $O(1)$
Descripción: devuelve un iterador al conjunto de campos claves de la tabla indicada.

InsertarEntrada(**in** r: `registro`, **in** t: `tabla`, **in/out** b: `base`)
Pre $\equiv$ {$t \in$ tablas(b) $\wedge_{\text{L}}$ puedoInsertar?(r, t) $\wedge$ $b = b_0$}
Post $\equiv$ {$b =_{\text{obs}}$ insertarEntrada(r, t, b_0)}
Complejidad: $O\big(log\ n + |L| * \#tablas(b)\big)$, donde L es el dato string más largo de r y n es la cantidad de registros en la tabla.
Descripción: inserta un registro en una tabla de la base de datos.

Borrar(**in** cr: `registro`, **in** t: `tabla`, **in/out** b: `base`)
Pre $\equiv$ {#campos(cr) $= 1 \wedge t \in$ tablas(b) $\wedge$ $b = b_0$}
Post $\equiv$ {$b =_{\text{obs}}$ borrar(cr, t, b_0)}
Complejidad:
Campo indexado $\Rightarrow$ $O\big(log\ n + |L| * \#tablas(b)\big)$
Campo no indexado $\Rightarrow$ $O\big(|L| * (n + \#tablas(b))\big)$, donde L es el dato string más largo de cr y n es la cantidad de registros en la tabla.
Descripción: borra todos los registros que coincidan con el campo del registro cr en la tabla t.

GenerarVistaJoin(**in** t_1: `string`, **in** t_2: `string`, **in** c: `campo`, **in/out** b: `base`)
Pre $\equiv$ {$t_1 \neq_{\text{obs}} t_2 \wedge \{t1, t2\} \subseteq$ tablas(b) $\wedge_{\text{L}}$ $c \in$ claves(dameTabla(t_1, b)) $\wedge$ $c \in$ claves(dameTabla(t_2, b)) $\wedge$ $\neg$(hayJoin?(t_1, t_2, b)) $\wedge$ tipoCampo(c, t_1) = tipoCampo(c, t_2) $\wedge$ $b = b_0$}
Post $\equiv$ {$b =_{\text{obs}}$ generarVistaJoin(t_1, t_2, c, b_0)}
Complejidad:
c cualquier tipo indexado en t_1 y t_2 $\Rightarrow$ $O\big((n+m) * \big(L + log(n+m)\big)\big)$
c cualquier tipo no indexado $\Rightarrow$ $O\big((n+m)\big(L + log(n+m)\big) + L * n * m\big)$
Donde n = #registros(t_1), m = #registros(t_2) y L el dato string más largo de cualquiera de las dos tablas.
Descripción: crea un join entre dos tablas de la base de datos.

BorrarJoin(**in** t_1: `string`, **in** t_2: `string`, **in/out** b: `base`)
Pre $\equiv$ {hayJoin?(t_1, t_2, b) $\wedge$ $b = b_0$}
Post $\equiv$ {$b =_{\text{obs}}$ borrarJoin(t_1, t_2, b_0)}
Complejidad: $O(1)$
Descripción: elimina el join entre dos tablas.

Tablas(**in** b: `base`) $\rightarrow$ *res* : `itBi(string)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ crearIt(tablas(b))}
Complejidad: $O(1)$
Descripción: se obtienen todas las tablas de la base de datos.

DameTabla(**in** s: `string`, **in** b: `base`) → res : `tabla`
Pre ≡ {$s \in$ tablas(b)}
Post ≡ {$res =_{obs}$ dameTabla(s)}
Complejidad: $O(1)$
Descripción: dado un nombre, devuelve la tabla con ese nombre en la base de datos.

HayJoin?(**in** t_1 : `string`, **in** t_2 : `string`, **in** b: `base`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{obs}$ hayJoin?(t_1, t_2, b)}
Complejidad: $O(1)$
Descripción: devuelve `true` si hay un join entre los dos nombres de las tablas dados.

CampoJoin(**in** t_1 : `string`, **in** t_2 : `string`, **in** b: `base`) → res : `campo`
Pre ≡ {hayJoin?(t_1, t_2, b)}
Post ≡ {$res =_{obs}$ campoJoin(t_1, t_2, b)}
Complejidad: $O(1)$
Descripción: devuelve el campo que une al join entre las dos tablas.

VistaJoin(**in** t_1 : `string`, **in** t_2 : `string`, **in** b: `base`) → res : `itBi(registro)`
Pre ≡ {hayJoin?(t_1, t_2, b)}
Post ≡ {$res =_{obs}$ vistaJoin(t_1, t_2, b)}
Complejidad: ver en los algoritmos para los distintos casos.
Descripción: devuelve un iterador a los conjuntos del join (ya definido) entre las dos tablas.

TablaMaxima(**in** b: `base`) → res : `string`
Pre ≡ {#tablas(b) >0}
Post ≡ {$res =_{obs}$ tablaMaxima(b)}
Complejidad: $O(1)$
Descripción: evuelve el nombre de la tabla más accedida.

Representación

Representación de base

`base` **se representa con** `estr`

donde `estr` es `tupla` ⟨*tablaMasAccedida*: `puntero(string)`, *nombreATabla*: `diccString(tabla)`, *tablas*: `lista(string)`, *joinPorCampoNat*: `diccString(diccString(diccNat(itConj(registro))))`, *joinPorCampoString*: `diccString(diccString(diccString(itConj(registro))))`, *registrosDelJoin*: `diccString(diccString(conj(registro)))`, *hayJoin*: `diccString(diccString(tupla< campoJoin: campo, cambios: lista(tupla<reg: registro, agregar?: bool>)>))`⟩

donde `registro` es `diccString(dato)` y se explica con Registro.

Invariante de representación
1) Las claves de nombreATabla están en tablas, sus significados tienen su nombre y son todas las tablas de la lista e.tablas. Y la lista de e.tablas no tiene repetidos.
2) La tabla más accedida está en e.tablas y tiene más accesos que todas las demás.
3) Las claves de JoinPorCampo, hayJoin y registrosDelJoin son las tablas de e.tablas (y por (1), las de nombreATabla).
4) No hay tablas con joins con ellas mismas.
5) Los significados de una clave en las estructuras relacionadas a los joins son las mismas para cada estructura (son aquellas con las que comparten un join).
6) En JoinPorCampoNat, joinPorCampoString, registrosDelJoin y hayJoin, las claves son recíprocas.
7) Entre dos tablas solamente puede haber un único join.
8) El campo del join también es recíproco y es clave para los dos.
9) El campo del join en hayJoin es el que lo define en el diccionario según su tipo (que es el mismo tipo para ambas tablas).
10) Los significados de cada diccionario de joins tienen siguiente perteneciente a registros del join para las mismas claves.

11) Para cada registro en registros del join hay un iterador en alguno de los dos diccionarios (nat o string) con siguiente en él.

12) Para cada registro en registros, sus campos son la unión de los campos de las dos tablas.

13) Los iteradores de los diccionarios también son recíprocos entre las tablas.

14) Los registros del join también son recíprocos entre las tablas.

15) Los registros en la lista de cambios tienen por campos a los campos de la primer clave y, para cada última aparación de un registro en la lista, el bool agregar refleja si pertenece el registro o no a los registros de las primer clave.

$$
\begin{array}{l}
\text{Rep} : \text{estr} \longrightarrow \text{bool} \\
\text{Rep}(e) \equiv \text{true} \iff \\
\quad \left(\begin{array}{l} \textbf{1)} \ (\forall\, s : \text{string}) \ \Big(s \in \text{claves}(e.\text{nombreATabla}) \wedge_{\text{L}} \text{nombre}\big(\text{obtener}(s,\, e.\text{nombreATabla})\big) =_{\text{obs}} s \Big) \iff \\ \big(\text{esta?}(s,\, e.\text{tablas}) \wedge \text{sinRepetidos}(e.\text{tablas})\big) \end{array} \right) \wedge \\
\quad \left(\begin{array}{l} \textbf{2)} \ e.\text{tablaMasAccedida} \in \text{claves}(e.\text{nombreATabla}) \wedge_{\text{L}} \ \big(\forall\, n : \text{string},\, n \in \text{claves}(e.\text{nombreATabla})\big) \\ \text{cantAccesos}\big(\text{obtener}(n,\, e.\text{nombreATabla})\big) \leq \text{cantAccesos}\big(\text{obtener}(*e.\text{tablaMasAccedida},\, e.\text{nombreATabla})\big) \end{array} \right) \wedge \\
\quad \left(\begin{array}{l} \textbf{3)} \ \text{claves}(e.\text{joinPorCampoNat}) =_{\text{obs}} \text{claves}(e.\text{hayJoin}) \wedge \\ \text{claves}(e.\text{joinPorCampoString}) =_{\text{obs}} \text{claves}(e.\text{hayJoin}) \wedge \\ \text{claves}(e.\text{registrosDelJoin}) =_{\text{obs}} \text{claves}(e.\text{hayJoin}) \wedge \\ \text{claves}(e.\text{hayJoin}) =_{\text{obs}} \text{claves}(e.\text{nombreATabla}) \end{array} \right) \wedge \\
\quad \Big(\textbf{4)} \ \neg\big(\exists\, s : \text{string},\, s \in \text{claves}(e.\text{hayJoin})\big) \ \big(s \in \text{claves}\big(\text{obtener}(s,\, e.\text{hayJoin})\big)\big) \Big) \wedge \\
\quad \left(\begin{array}{l} \textbf{5)} \ \big(\forall\, n : \text{string},\, n \in \text{claves}(e.\text{hayJoin})\big) \\ \Big(\begin{array}{l} \text{claves}\big(\text{obtener}(n,\, e.\text{JoinPorCampoNat})\big) \cup \\ \text{claves}\big(\text{obtener}(n,\, e.\text{JoinPorCampoString})\big) \end{array} \Big) =_{\text{obs}} \text{claves}\big(\text{obtener}(n,\, e.\text{hayJoin})\big) \wedge \\ \text{claves}\big(\text{obtener}(n,\, e.\text{registrosDelJoin})\big) =_{\text{obs}} \text{claves}\big(\text{obtener}(n,\, e.\text{hayJoin})\big) \wedge \\ \text{claves}\big(\text{obtener}(n,\, e.\text{hayJoin})\big) =_{\text{obs}} \text{claves}\big(\text{obtener}(n,\, e.\text{nombreATabla})\big) \end{array} \right) \wedge \\
\quad \left(\begin{array}{l} \textbf{6)} \ \big(\forall\, s_1 : \text{string},\, s_1 \in \text{claves}(e.\text{hayJoin})\big) \\ \left(\begin{array}{l} \big(\forall\, s_2 : \text{string},\, s_2 \in \text{claves}(e.\text{hayJoin}) \wedge s_1 \neq_{\text{obs}} s_2\big) \\ \Big(s_1 \in \text{claves}\big(\text{obtener}(s_2,\, e.\text{hayJoin})\big) \iff s_2 \in \text{claves}\big(\text{obtener}(s_1,\, e.\text{hayJoin})\big) \Big) \end{array} \right) \end{array} \right) \wedge \\
\quad \left(\begin{array}{l} \textbf{7)} \ \big(\forall\, s_1 : \text{string},\, s_1 \in \text{claves}(e.\text{hayJoin})\big) \\ \big(\forall\, s_2 : \text{string},\, s_2 \in \text{claves}\big(\text{obtener}(s_1,\, e.\text{hayJoin})\big)\big) \\ \Big(\text{def?}\big(s_2,\, \text{obtener}(s_1,\, e.\text{joinPorCampoNat})\big) \ \Rightarrow \ \neg\text{def?}\big(s_2,\, \text{obtener}(s_1,\, e.\text{joinPorCampoString})\big) \Big) \wedge \\ \Big(\text{def?}\big(s_2,\, \text{obtener}(s_1,\, e.\text{joinPorCampoString})\big) \ \Rightarrow \ \neg\text{def?}\big(s_2,\, \text{obtener}(s_1,\, e.\text{joinPorCampoNat})\big) \Big) \end{array} \right) \wedge \\
\quad \left(\begin{array}{l} \textbf{8)} \ \big(\forall\, s_1 : \text{string},\, s_1 \in \text{claves}(e.\text{hayJoin})\big) \\ \big(\forall\, s_2 : \text{string},\, s_2 \in \text{claves}\big(\text{obtener}(s_1,\, e.\text{hayJoin})\big)\big) \\ \Big(\text{obtener}\big(s_2,\, \text{obtener}(s_1,\, e.\text{hayJoin})\big).\text{campo} =_{\text{obs}} \text{obtener}\big(s_1,\, \text{obtener}(s_2,\, e.\text{hayJoin})\big).\text{campo} \Big) \wedge_{\text{L}} \\ \Big(\text{obtener}\big(s_2,\, \text{obtener}(s_1,\, e.\text{hayJoin})\big).\text{campo} \in \text{claves}\big(\text{obtener}(s_1,\, \text{e.nombreATabla})\big) \Big) \wedge \\ \Big(\text{obtener}\big(s_2,\, \text{obtener}(s_1,\, e.\text{hayJoin})\big).\text{campo} \in \text{claves}\big(\text{obtener}(s_2,\, e.\text{nombreATabla})\big) \Big) \end{array} \right) \wedge \\
\quad \left(\begin{array}{l} \textbf{9)} \ \big(\forall\, s_1 : \text{string},\, s_1 \in \text{claves}(e.\text{hayJoin})\big) \\ \big(\forall\, s_2 : \text{string},\, s_2 \in \text{claves}\big(\text{obtener}(s_1,\, e.\text{hayJoin})\big)\big) \\ \left(\begin{array}{l} \text{tipoCampo}\Big(\text{obtener}\big(s_2,\, \text{obtener}(s_1,\, e.\text{hayJoin})\big).\text{campo},\, \text{obtener}\big(e.\text{nombreATabla}(s_1)\big)\Big) =_{\text{obs}} \\ \text{tipoCampo}\Big(\text{obtener}\big(s_2,\, \text{obtener}(s_1,\, e.\text{hayJoin})\big).\text{campo},\, \text{obtener}\big(\text{e.nombreATabla}(s_2)\big)\Big) \wedge_{\text{L}} \\ \left(\begin{array}{l} \left(\begin{array}{l} \Big(\text{tipoCampo}\Big(\text{obtener}\big(s_2,\, \text{obtener}(s_1,\, e.\text{hayJoin})\big).\text{campo},\, \text{obtener}\big(\text{e.nombreATabla}(s_2)\big)\Big) \Big) \ \Rightarrow \\ \text{def?}\big(s_2,\, \text{obtener}(s_1,\, e.\text{joinPorCampoNat})\big) \end{array} \right) \wedge \\ \left(\begin{array}{l} \Big(\neg\text{tipoCampo}\Big(\text{obtener}\big(s_2,\, \text{obtener}(s_1,\, e.\text{hayJoin})\big).\text{campo},\, \text{obtener}\big(\text{e.nombreATabla}(s_2)\big)\Big) \Big) \ \Rightarrow \\ \text{def?}\big(s_2,\, \text{obtener}(s_1,\, e.\text{joinPorCampoString})\big) \end{array} \right) \end{array} \right) \end{array} \right) \wedge
\end{array}
$$

$$
\begin{pmatrix}
\textbf{10)}\ (\forall\ s_1 : \text{string},\ s_1 \in \text{claves}(e.\text{joinPorCampoNat})) \\
\left(\begin{pmatrix}(\forall\ s_2 : \text{string},\ s_2 \in \text{claves}\big(\text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)) \\
\left(\begin{pmatrix}\big(\forall\ n : \text{nat},\ \text{def?}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)\big)\big) \\
\begin{pmatrix}\text{siguiente}\Big(\text{obtener}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)\big)\Big) \in \\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{registrosDelJoin})\big)\end{pmatrix}\end{pmatrix}\right)\end{pmatrix} \wedge \\
(\forall\ s_1 : \text{string},\ s_1 \in \text{claves}(e.\text{joinPorCampoString})) \\
\left(\begin{pmatrix}(\forall\ s_2 : \text{string},\ s_2 \in \text{claves}\big(\text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)) \\
\left(\begin{pmatrix}\big(\forall\ n : \text{nat},\ \text{def?}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)\big)\big) \\
\begin{pmatrix}\text{siguiente}\Big(\text{obtener}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)\big)\Big) \in \\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{registrosDelJoin})\big)\end{pmatrix}\end{pmatrix}\right)\end{pmatrix}
\end{pmatrix} \wedge
$$

$$
\begin{pmatrix}
\textbf{11)}\ (\forall\ s_1 : \text{string},\ s_1 \in \text{claves}(e.\text{registrosDelJoin})) \\
\left(\begin{pmatrix}(\forall\ s_2 : \text{string},\ s_2 \in \text{claves}\big(\text{obtener}(s_1,\ e.\text{registrosDelJoin})\big)) \\
\left(\begin{pmatrix}(\forall\ r : \text{registro},\ r \in \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{registrosDelJoin})\big)) \\
\left(\begin{pmatrix}\text{def?}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)\ \Rightarrow \\
\left(\begin{pmatrix}\big(\exists\ n : \text{nat},\ \text{def?}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)\big)\big) \\ \text{siguiente}\Big(\text{obtener}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)\big)\Big) =_{\text{obs}} r\end{pmatrix}\right) \wedge \\
\left(\begin{pmatrix}\text{def?}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)\ \Rightarrow \\
\left(\begin{pmatrix}\big(\exists\ n : \text{string},\ \text{def?}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)\big)\big) \\ \text{siguiente}\Big(\text{obtener}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)\big)\Big) =_{\text{obs}} r\end{pmatrix}\right)\end{pmatrix}\right)\end{pmatrix}\right)\end{pmatrix}\right)\end{pmatrix} \wedge
$$

$$
\begin{pmatrix}
\textbf{12)}\ (\forall\ s_1 : \text{string},\ s_1 \in \text{claves}(e.\text{registrosDelJoin})) \\
\left(\begin{pmatrix}(\forall\ s_2 : \text{string},\ s_2 \in \text{claves}\big(\text{obtener}(s_1,\ e.\text{registrosDelJoin})\big)) \\
\left(\begin{pmatrix}(\forall\ r : \text{registro},\ r \in \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{registrosDelJoin})\big)) \\
\big(\text{campos}(r) =_{\text{obs}} \big(\text{campos}\big(\text{obtener}(s_1,\ e.\text{nombreATabla})\big) \cup \text{campos}\big(\text{obtener}(s_2,\ e.\text{nombreATabla})\big)\big)\big)\end{pmatrix}\right)\end{pmatrix}\right)\end{pmatrix} \wedge
$$

$$
\begin{pmatrix}
\textbf{13)}\ (\forall\ s_1 : \text{string},\ s_1 \in \text{claves}(e.\text{joinPorCampoString})) \\
\left(\begin{pmatrix}(\forall\ s_2 : \text{string},\ s_2 \in \text{claves}\big(\text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)) \\
\left(\begin{pmatrix}\big(\forall\ n : \text{string},\ \text{def?}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)\big)\big) \\
\begin{pmatrix}\text{def?}\big(n,\ \text{obtener}\big(s_1,\ \text{obtener}(s_2,\ e.\text{joinPorCampoString})\big)\big) \wedge_{\text{L}} \\ \text{obtener}\big(n,\ \text{obtener}\big(s_1,\ \text{obtener}(s_2,\ e.\text{joinPorCampoString})\big)\big) =_{\text{obs}} \\ \text{obtener}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoString})\big)\big)\end{pmatrix}\end{pmatrix}\right)\end{pmatrix} \wedge \\
(\forall\ s_1 : \text{string},\ s_1 \in \text{claves}(e.\text{joinPorCampoNat})) \\
\left(\begin{pmatrix}(\forall\ s_2 : \text{string},\ s_2 \in \text{claves}\big(\text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)) \\
\left(\begin{pmatrix}\big(\forall\ n : \text{nat},\ \text{def?}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)\big)\big) \\
\begin{pmatrix}\text{def?}\big(n,\ \text{obtener}\big(s_1,\ \text{obtener}(s_2,\ e.\text{joinPorCampoNat})\big)\big) \wedge_{\text{L}} \\ \text{obtener}\big(n,\ \text{obtener}\big(s_1,\ \text{obtener}(s_2,\ e.\text{joinPorCampoNat})\big)\big) =_{\text{obs}} \\ \text{obtener}\big(n,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{joinPorCampoNat})\big)\big)\end{pmatrix}\end{pmatrix}\right)\end{pmatrix}
\end{pmatrix} \wedge
$$

$$
\begin{pmatrix}
\textbf{14)}\ (\forall\ s_1 : \text{string},\ s_1 \in \text{claves}(e.\text{registrosDelJoin})) \\
\left(\begin{pmatrix}(\forall\ s_2 : \text{string},\ s_2 \in \text{claves}\big(\text{obtener}(s_1,\ e.\text{registrosDelJoin})\big)) \\
\left(\begin{pmatrix}(\forall\ r : \text{registro}) \\
\begin{pmatrix}r \in \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{registrosDelJoin})\big) \iff \\ r \in \text{obtener}\big(s_1,\ \text{obtener}(s_2,\ e.\text{registrosDelJoin})\big)\end{pmatrix}\end{pmatrix}\right)\end{pmatrix}\right)\end{pmatrix} \wedge
$$

$$
\begin{pmatrix}
\textbf{15)}\ (\forall\ s_1 : \text{string},\ s_1 \in \text{claves}(e.\text{hayJoin})) \\
\left(\begin{pmatrix}(\forall\ s_2 : \text{string},\ s_2 \in \text{claves}\big(\text{obtener}(s_1,\ e.\text{hayJoin})\big)) \\
\left(\begin{pmatrix}\Big(\forall\ t : \text{tupla}(\text{registro},\ \text{bool}),\ \text{esta?}\Big(t,\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{hayJoin})\big).\text{cambios}\Big)\Big) \\
\begin{pmatrix}\text{campos}\big(\Pi_1(t)\big) =_{\text{obs}} \text{campos}\big(\text{obtener}(s_1,\ e.\text{nombreATabla})\big) \wedge \\ \text{estaAgregado?}\Big(\Pi_1(t),\ \text{obtener}\big(s_2,\ \text{obtener}(s_1,\ e.\text{hayJoin})\big).\text{cambios}\Big) \iff \\ \Pi_1(t) \in \text{registros}\big(\text{obtener}(s_1,\ e.\text{nombreATabla})\big)\end{pmatrix}\end{pmatrix}\right)\end{pmatrix}\right)\end{pmatrix}
$$

$\text{sinRepetidos} : \text{secu}(\text{string}) \longrightarrow \text{bool}$
$\text{sinRepetidos}(ls) \equiv \textbf{if}\ \text{vacia?}(ls)\ \textbf{then}\ \text{true}\ \textbf{else}\ \neg\text{esta?}\big(\text{prim}(ls),\ \text{fin}(ls)\big) \wedge_{\text{L}} \text{sinRepetidos}\big(\text{fin}(ls)\big)\ \textbf{fi}$

estaAgregado? : registro r × secu(tupla(registro,bool)) s $\longrightarrow$ bool
$\{\text{esta?}(<r, \text{true}>, s) \vee \text{esta?}(<r, \text{false}>, s)\}$
estaAgregado?(t, s) $\equiv$ **if** $\Pi_1(\text{ult}(s)) =_{\text{obs}} r$ **then** $\Pi_2(\text{ult}(s))$ **else** estaAgregado?$(t, \text{com}(s))$ **fi**

Abs : estr e $\longrightarrow$ base $\{\text{Rep}(e)\}$
Abs(e) $=_{\text{obs}}$ b: base | tablas(b) $=_{\text{obs}}$ $(\text{claves}(e.\text{nombreATabla}))$ $\wedge$
$\Big((\forall\, s : \text{string})\ (s \in \text{claves}(e.\text{nombreATabla}))\ \Rightarrow\ ((\text{dameTabla}(s, b) =_{\text{obs}} \text{obtener}(s, e.\text{nombreATabla}))\Big)$
$\wedge\ \Big((\forall\ s_1,\ s_2 : \text{string})\ (s_1 \in \text{claves}(e.\text{nombreATabla}) \wedge s_2 \in \text{claves}(e.\text{nombreATabla}))\ \Rightarrow$
$(\text{hayJoin?}(s_1,\ s_1,\ b) =_{\text{obs}} \text{def?}(s_2,\ \text{obtener}(s_1,\ e.\text{hayJoin}))) \wedge_{\text{L}} \text{campoJoin}(s_1,\ s_2,\ b) =_{\text{obs}}$
$\Pi_1(\text{obtener}(s_2,\ (\text{obtener}(s_1,\ e.\text{hayJoin}))))\ \Big)$

Algoritmos

Algoritmos de base

---

iNuevaBDD() $\rightarrow$ *res* : base
- $res.tablaMasAccedida \leftarrow NULL$ ▷ $O(1)$
- $res.nombreATabla \leftarrow Vacio()$ ▷ $O(1)$
- $res.tablas \leftarrow Vacia()$ ▷ $O(1)$
- $res.hayJoin \leftarrow Vacio()$ ▷ $O(1)$
- $res.joinPorCampoNat \leftarrow Vacio()$ ▷ $O(1)$
- $res.joinPorCampoString \leftarrow Vacio()$ ▷ $O(1)$
- $res.registrosDelJoin \leftarrow Vacio()$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Todas las funciones llamadas tienen complejidad $O(1)$.

---

iAgregarTabla(**in** t: `tabla`, **in/out** e: `estr`)
// Si no hay tabla más accedida o la tabla que agregue tiene más accesos que la más accedida de la bdd...
if $e.tablaMasAccedida = NULL \vee_{\text{L}} CantidadDeAccesos(nombreATabla(*e.tablaMasAccedida)) < CantidadDeAccesos(t)$ **then** ▷ $O(1)$
  $e.tablaMasAccedida \leftarrow \&Nombre(t)$ ▷ $O(1)$
end if

// Agrego la tabla a todos lados
- $Definir(Nombre(t), t, e.nombreATabla)$ ▷ $O(|nombre(t)|) = O(1)$
- $AgregarAtras(nombre(t), e.tablas)$ ▷ $O(1)$
- $Definir(Nombre(t), Vacio(), e.hayJoin)$ ▷ $O(|nombre(t)|) = O(1)$
- $Definir(Nombre(t), Vacio(), e.joinPorCampoNat)$ ▷ $O(|nombre(t)|) = O(1)$
- $Definir(Nombre(t), Vacio(), e.joinPorCampoString)$ ▷ $O(|nombre(t)|) = O(1)$
- $Definir(Nombre(t), Vacio(), e.registrosDelJoin)$ ▷ $O(|nombre(t)|) = O(1)$

Complejidad: $O(1)$
Justificación: Por estar acotados los nombres de las tablas, Definir en diccString con nombres por clave se hace en $O(1)$.

---

iInsertarEntrada(**in** r : `registro`, **in** t : `tabla`, **in**/**out** e : `estr`)

```
AgregarRegistro(r, t)                                  ▷ O(|L| + log n) indexado / O(|L|) no indexado
// Me fijo si cambió la tabla más accedida
tabMax ← e.nombreATabla(*(e.tablaMasAccedida))                                          ▷ O(1)
if CantidadDeAccesos(t) > CantidadDeAccesos(tabMax) then                                ▷ O(1)
    e.tablaMasAccedida ← &Nombre(t)                                                     ▷ O(1)
end if

// Lo tengo que agregar a cambios con las tablas que tenga join
iter ← VistaDicc(Obtener(Nombre(t), e.hayJoin))                                         ▷ O(1)
while HaySiguiente?(iter) do                                                            ▷ O(#tablas * L)
    AgregarAtras(< r, true >, Siguiente(iter).significado.cambios)
                                                  ▷ O(copy(r)) = O(#campos * dato mas costoso) = O(L)
    Avanzar(iter)                                                                       ▷ O(1)
end while
```

Complejidad:

Campo indexado $\Rightarrow$ $O(|L|+log\ n+\#tablas*|L|) = O(log\ n+|L|*(\#tablas+1)) = O(log\ n+|L|*\#tablas(b))$

Campo no indexado $\Rightarrow$ $O(|L| * \#tablas(b))$

Donde L es el dato string más largo de r y n es la cantidad de registros en la tabla.

Justificación:

Por interfaz de Tabla, agregar el registro a la tabla indicada cuesta $O(|L|+log\ n))$ si hay algún campo indexado, y si no, $O(|L|)$.

Obtener la tabla más accedida a partir de su nombre cuesta $O(Nombre\ mas\ largo\ de\ tabla\ de\ la\ base)$, pero como están acotadas en longitud de nombre eso equivale a $O(1)$.

Las operaciones & y * para el tipo primitivo puntero cuestan $O(1)$.

El puntero al nombre de la tabla más accedida se asigna por referencia en O(1).

VistaDicc exporta complejidad $O(1)$.

En el peor caso se agrega por copia el registro a la lista de cambios de todas las demás tablas (asumiendo que tiene joins con todas). Eso equivale a $O(\#campos * dato\ mas\ costoso\ de\ copiar)$ por cada inserción, pero como los registros tienen cantidad de campos acotados, se reduce la complejidad a $O(L)$. Por lo tanto el ciclo cuesta $O(L * \#tablas(b))$.

Entonces si hay algún campo indexado nos queda $O(|L| * \#Tablas(b) + log\ n)$ y si no hay un campo indexado nos queda $O(|L| * \#tablas(b))$

iBorrar(**in** cr : `registro`, **in** t : `tabla`, **in**/**out** e : `estr`)

$BorrarRegistro(cr, t)$ ▷ *Campo indexado* $\Rightarrow O(log\ n + L)$ / *Campo no indexado* $\Rightarrow O(n * |L|)$
$tabMax \leftarrow e.nombreATabla(*(e.tablaMasAccedida))$ ▷ $O(1)$
if $CantidadDeAccesos(t) > CantidadDeAccesos(tabMax)$ **then** ▷ $O(1)$
  $e.tablaMasAccedida \leftarrow \&Nombre(t)$ ▷ $O(1)$
end if
$iter \leftarrow VistaDicc(Obtener(Nombre(t), e.hayJoin))$ ▷ $O(1)$
while **do**$HaySiguiente?(iter)$ ▷ $O(\#tablas * |L|)$
  $AgregarAtras(< cr, false >, Siguiente(iter).significado.cambios)$
  ▷ $O(copy(r)) = O(\#campos * dato\ mas\ costoso) = O(L)$
  $Avanzar(iter)$ ▷ $O(1)$
end while

Complejidad:

Campo indexado $\Rightarrow O(log\ n + |L| + \#tablas * |L|) = O(log\ n + |L| * (\#tablas + 1)) = O(log\ n + |L| * \#tablas)$
Campo no indexado $\Rightarrow O(n * |L| + \#tablas * |L|) = O(|L| * (n + \#tablas))$

Justificación:

Obtener la tabla más accedida a partir de su nombre cuesta $O(Nombre\ mas\ largo\ de\ tabla\ de\ la\ base)$, pero como están acotadas en longitud de nombre, eso equivale a $O(1)$.

Las operaciones & y * para el tipo primitivo puntero cuestan $O(1)$.

El puntero al nombre de la tabla más accedida se asigna por referencia en $O(1)$.

VistaDicc exporta complejidad $O(1)$.

En el peor caso se agrega por copia el registro a la lista de cambios de todas las demás tablas (asumiendo que tiene joins con todas). Eso equivale a $O(\#campos * dato\ mas\ costoso\ de\ copiar)$ por cada inserción, pero como los registros tienen cantidad de campos acotados, se reduce la complejidad a $O(L)$. Por lo tanto el ciclo cuesta $O(L * \#Tablas(b))$.

BorrarRegistro exporta complejidad distinta dependiendo de si hay índice sobre el campo criterio y se suma al resto diferenciando cada caso.

iGenerarVistaJoin(**in** t_1 : `string`, **in** t_2 : `string`, **in** c : `campo`, **in/out** e : `estr`) → res : itConj(registro)

// Creo en el diccionario hayJoin de cada tabla la otra tabla.
$aux \leftarrow < c, Vacio() >$ ▷ $O(1)$
$Definir(t_2, aux, Obtener(t_1, e.hayJoin))$ ▷ $O(|maximo\ nombre\ de\ tabla|) = O(1)$
$Definir(t_1, aux, Obtener(t_2, e.hayJoin))$ ▷ $O(|maximo\ nombre\ de\ tabla|) = O(1)$

$Definir(t_2, Vacio(), Obtener(t_1, e.registrosDelJoin))$ ▷ $O(|maximo\ nombre\ de\ tabla|) = O(1)$
$Definir(t_1, Vacio(), Obtener(t_2, e.registrosDelJoin))$ ▷ $O(|maximo\ nombre\ de\ tabla|) = O(1)$

// Si es nat el campoJoin...
if $TipoCampo(c, t1)$ **then** ▷ $O(1)$ Defino en cada diccionario de joinPorCampoNat la otra tabla.
$Definir(t_2, Vacio(), Obtener(t_1, e.joinPorCampoNat))$ ▷ $O(1)$
$Definir(t_1, Vacio(), Obtener(t_2, e.joinPorCampoNat))$ ▷ $O(1)$

// Itero sobre los registros de t1 buscando en t2 por el registro (es único por ser clave el campo, req) que comparta ese dato (si lo hay), los mergeamos y definimos en el diccionario de t1 a t2 (y viceversa)

$it \leftarrow CrearIt(Registros(t_1))$ ▷ $O(1)$
while $HaySiguiente(it)$ **do** ▷ $O(n * ...)$
$d \leftarrow Obtener(c, Siguiente(it))$ ▷ $O(1)$
$coincis \leftarrow Buscar(c, d, t_2)$ ▷ *Si esta indexado* $\Rightarrow O(log\ m + |L|)$ / *Si no esta indexado* $\Rightarrow O(m * |L|)$
if not $Vacia?(coincis)$ **then** ▷ $O(L)$
$regMergeado \leftarrow Merge(Siguiente(it), Primero(coincis))$ ▷ $O(L)$
// Agrego a los dos conjuntos de registros (son iguales)
$iter_1 \leftarrow AgregarRapido(regMergeado, Obtener(t_2, Obtener(t_1, e.registrosDelJoin)))$ ▷ $O(L)$
$iter_2 \leftarrow AgregarRapido(regMergeado, Obtener(t_1, Obtener(t_2, e.registrosDelJoin)))$ ▷ $O(L)$
$n \leftarrow ValorNat(d)$ ▷ $O(1)$
$Definir(n, iter_1, Obtener(t_2, Obtener(t_1, e.joinPorCampoNat)))$ ▷ $O(log\ n)$
$Definir(n, iter_2, Obtener(t_1, Obtener(t_2, e.joinPorCampoNat)))$ ▷ $O(log\ n)$
end if
$Avanzar(it)$ ▷ $O(1)$
end while

// Itero sobre registros de t2 igual que arriba pero además de preguntando si ya los definimos antes

$it \leftarrow CrearIt(Registros(t_2))$ ▷ $O(1)$
while $HaySiguiente(it)$ **do** ▷ $O(m * ...)$
$d \leftarrow Obtener(c, Siguiente(it))$ ▷ $O(1)$
$coincis \leftarrow Buscar(c, d, t_1)$ ▷ *Si esta indexado* $\Rightarrow O(log\ n + |L|)$ / *Si no esta indexado* $\Rightarrow O(n * |L|)$
$n \leftarrow ValorNat(d)$
if not $Vacia?(coincis)$ **and not** $Def?(n, Obtener(t_1, Obtener(t_2, e.joinPorCampoNat)))$ **then**
▷ $O(log(n + m))$
$regMergeado \leftarrow Merge(Siguiente(it), Primero(coincis))$ ▷ $O(L)$
// Agrego a los dos conjuntos de registros (son iguales)
$iter_1 \leftarrow AgregarRapido(regMergeado, Obtener(t_2, Obtener(t_1, e.registrosDelJoin)))$ ▷ $O(L)$
$iter_2 \leftarrow AgregarRapido(regMergeado, Obtener(t_1, Obtener(t_2, e.registrosDelJoin)))$ ▷ $O(L)$
$Definir(n, iter_1, Obtener(t_2, Obtener(t_1, e.joinPorCampoNat)))$ ▷ $O(log\ n + m)$
$Definir(n, iter_2, Obtener(t_1, Obtener(t_2, e.joinPorCampoNat)))$ ▷ $O(log\ n + m)$
end if
$Avanzar(it)$ ▷ $O(1)$
end while

else
// mismo que arriba pero para joinPorCampoString
$Definir(t_2, Vacio(), Obtener(t_1, e.joinPorCampoString))$ ▷ $O(1)$
$Definir(t_1, Vacio(), Obtener(t_2, e.joinPorCampoString))$ ▷ $O(1)$

// Itero sobre registros de t1 buscando en t2 por cada clave el registro (es único por ser clave el campo, req) que comparta ese dato (si lo hay) y los mergeamos y definimos en el diccionario de t1 a t2 (y viceversa)

$it \leftarrow CrearIt(Registros(t_1))$ ▷ $O(1)$
while $HaySiguiente(it)$ **do** ▷ $O(n * ...)$
  $d \leftarrow Obtener\big(c, Siguiente(it)\big)$ ▷ $O(1)$
  $coincis \leftarrow Buscar(c, d, t_2)$ ▷ *Si esta indexado* $\Rightarrow O(|L|)$ / *Si no esta indexado* $\Rightarrow O(m * |L|)$
  if not $Vacia?(coincis)$ **then** ▷ $O(1)$
    $regMergeado \leftarrow Merge\big(Siguiente(it), Primero(coincis)\big)$ ▷ $O(L)$
    $s \leftarrow ValorStriing(d)$ ▷ $O(L)$
    // Agrego a los dos conjuntos de registros (son iguales)
    $iter_1 \leftarrow AgregarRapido\big(regMergeado, Obtener\big(t_2, Obtener(t_1, e.registrosDelJoin)\big)\big)$ ▷ $O(L)$
    $iter_2 \leftarrow AgregarRapido\big(regMergeado, Obtener\big(t_1, Obtener(t_2, e.registrosDelJoin)\big)\big)$ ▷ $O(L)$
    $Definir\big(s, iter_1, Obtener\big(t_2, Obtener(t_1, e.joinPorCampoString)\big)\big)$ ▷ $O(L)$
    $Definir\big(s, iter_2, Obtener\big(t_1, Obtener(t_2, e.joinPorCampoString)\big)\big)$ ▷ $O(L)$
  end if
  $Avanzar(it)$ ▷ $O(1)$
end while

// Itero sobre registros de t2 igual que arriba pero además de preguntando si ya los definimos antes

$it \leftarrow CrearIt(Registros(t_2))$ ▷ $O(1)$
while $HaySiguiente(it)$ **do** ▷ $O(m * ...)$
  $d \leftarrow Obtener\big(c, Siguiente(it)\big)$ ▷ $O(1)$
  $coincis \leftarrow Buscar(c, d, t_1)$ ▷ *Si esta indexado* $\Rightarrow O(|L|)$ / *Si no esta indexado* $\Rightarrow O(n * |L|)$
  $s \leftarrow ValorString(d)$
  if not $Vacia?(coincis)$ **and not** $Def?\big(s, Obtener\big(t_1, Obtener(t_2, e.joinPorCampoString)\big)\big)$ **then**
    ▷ $O(L)$
    $regMergeado \leftarrow Merge\big(Siguiente(it), Primero(coincis)\big)$ ▷ $O(L)$
    // Agrego a los dos conjuntos de registros (son iguales)
    $iter_1 \leftarrow AgregarRapido\big(regMergeado, Obtener\big(t_2, Obtener(t_1, e.registrosDelJoin)\big)\big)$ ▷ $O(L)$
    $iter_2 \leftarrow AgregarRapido\big(regMergeado, Obtener\big(t_1, Obtener(t_2, e.registrosDelJoin)\big)\big)$ ▷ $O(L)$
    $Definir\big(s, iter_1, Obtener\big(t_2, Obtener(t_1, e.joinPorCampoString)\big)\big)$ ▷ $O(L)$
    $Definir\big(s, iter_2, Obtener\big(t_1, Obtener(t_2, e.joinPorCampoString)\big)\big)$ ▷ $O(L)$
  end if
  $Avanzar(it)$ ▷ $O(1)$
end while

end if
$res \leftarrow CrearIt\big(Obtener\big(t_2, Obtener(t_1, e.registrosDelJoin)\big)\big)$

Complejidad:

c tipo nat indexado en t_1 y t_2 $\Rightarrow$
$O\big(n * (L + logn + logm)\big) + O\big(m * (L + logn + log(n + m)\big)$
$= O\big(n * (L + logn + logm)\big) + O\big(m * \big(L + log(n + m)\big)\big)$
$= O\big(n * \big(L + log(n + m) + log(n + m)\big)\big) + O\big(m * \big(L + log(n + m)\big)\big)$
$= O\big(n * \big(L + log(n + m)\big)\big) + O\big(m * \big(L + log(n + m)\big)\big)$
$= O\big((n + m) * \big(L + log(n + m)\big)\big)$

c tipo nat no indexado $\Rightarrow$
$O\big(n * (L + logn + m * L)\big) + O\big(m * (L + n * L + log(n + m))\big)$
$= O\big(L(n + m) + n(logn + m * L) + m\big(log(n + m) + n * L\big)\big)$
$= O\big(L(n + m) + n\big(log(n + m) + m * L\big) + m\big(log(n + m) + n * L\big)\big)$
$= O\big(\big(L + log(n + m)\big)(n + m) + n * m * L + n * m * L\big)$
$= O\big((n + m)\big(L + log(n + m)\big) + n * m * L\big)$

c tipo string indexado en t_1 y t_2 $\Rightarrow$ $O(n * L + m * L) = O\big(L * (n + m)\big)$
c tipo string no indexado $\Rightarrow$ $O\big(n * (m * L + L)\big) + O\big(m * (n * L + L)\big) = O\big(L * (m + n + n * m + n * m)\big) = O\big(L * (n + m + m * n)\big) = O(L * m * n)$

$\Rightarrow$ c cualquier tipo indexado en t_1 y t_2 $\Rightarrow$
$O\big(max\{(n+m)*(L+log(n+m)), L*(n+m)\}\big) = O\big((n+m)*\big(L+log(n+m)\big)\big)$

$\Rightarrow$ c cualquier tipo no indexado $\Rightarrow$
$O\big(max\{(n+m)(L+log(n+m))+L*n*m, L*m*n\}\big) = O\big((n+m)\big(L+log(n+m)\big)+L*n*m\big)$
Justificación: Definir el campo y los dos diccionarios vacíos para cada estructura es $O(1)$.
Obtener un significado en los diccString que tienen nombres de campos por claves también es $O(1)$ porque están acotados en longitud.

Caso nat: Se recorren los n registros de t_1 buscando coincidencias en t_2 para el campo c (si c es indexado en t_2 entonces esto se realiza en tiempo promedio logarítmico sobre la cantidad m de registros de t_2, si no es lineal con el agregado de copiar coincidencias considerando peor caso recorrer todos los registros).
Si hay coincidencias se deben mergear ambos registros en $O(L)$, siendo L el dato string más largo de cualquiera de las dos tablas, y luego definir el merge en las dos estructuras que almacenan registros de joins en común con complejidad Copy(merge) = $O(L)$ ya que al tener campos acotados sólo es significativo el costo de copiar una cantidad acotada de veces el dato más largo.
Se define para cada registro mergeado un iterador en cada diccNat de la estructura de joinPorCampoNat (como solamente se agregaron iteradores de registros de t_1 hasta ahora, log n peor caso).
Luego se aplica el mismo procedimiento para los m registros de t_2, pero esta vez considerando que no hayan sido agregados en el paso anterior, por lo que se agrega una búsqueda sobre la estructura de registros de joins (peor caso contiene todos los registros de ambas tablas: $O(log(n+m))$.

Caso string: Similar al caso nat pero considerando que las búsquedas e inserciones se realizan en tiempo de orden L (nuevamente, el dato string más largo de cualquiera de las dos tablas).

---

iBorrarJoin(**in** t_1 : `string`, **in** t_2 : `string`, **in**/**out** e : `estr`)

$Borrar\big(t_2, Obtener(t_1, e.hayJoin)\big)$	▷ $O(1)$
$Borrar\big(t_1, Obtener(t_2, e.hayJoin)\big)$	▷ $O(1)$
$Borrar\big(t_2, Obtener(t_1, e.registrosDelJoin)\big)$	▷ $O(1)$
$Borrar\big(t_1, Obtener(t_2, e.registrosDelJoin)\big)$	▷ $O(1)$
if $Def?\big(t_2, Obtener(t_1, e.joinPorCampoNat)\big)$ **then**	▷ $O(1)$
$Borrar\big(t_2, Obtener(t_1, e.joinPorCampoNat)\big)$	▷ $O(1)$
$Borrar\big(t_1, Obtener(t_2, e.joinPorCampoNat)\big)$	▷ $O(1)$
else	
$Borrar\big(t_2, Obtener(t_1, joinPorCampoString)\big)$	▷ $O(1)$
$Borrar\big(t_1, Obtener(t_2, joinPorCampoString)\big)$	▷ $O(1)$
end if	

Complejidad: $O(1)$
Justificación: Todas las operaciones de buscar y borrar en diccString se hacen en el orden del largo del máximo nombre de todas las tablas; al estar acotados estos nombres, estas operaciones se resuelven en $O(1)$.

---

iTablas(**in** e : `estr`) $\rightarrow$ res : itLista(string)

$res \leftarrow crearIt(e.tablas)$	▷ $O(1)$

Complejidad: $O(1)$
Justificación: Crear un iterador de una lista tiene complejidad $O(1)$.

---

iDameTabla(**in** s: `string`, **in** e: `estr`) $\rightarrow$ res : tabla

$res \leftarrow Obtener(s, e.nombreATabla)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Los nombres de las tablas están acotados, por lo tanto, buscar en un diccString con nombres por claves es $O(1)$.

---

iHayJoin?(**in** s_1: `string`, **in** s_2: `string`, **in** e: `estr`) $\rightarrow$ res : bool

$res \leftarrow Def?\big(s_2, Obtener(s_1, e.hayJoin)\big)$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Los nombres de las tablas están acotados, por lo tanto, buscar en un diccString con nombres por claves es $O(1)$.

---

iCampoJoin(**in** s_1: `string`, **in** s_2: `string`, **in** e: `estr`) $\rightarrow$ res : campo

$res \leftarrow \big(Obtener\big(s_2, Obtener(s_1, e.hayJoin)\big)\big).campoJoin$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Los nombres de las tablas están acotados, por lo tanto, buscar en un diccString con nombres por claves es $O(1)$.

MERGE(**in** r_1: `registro`, **in** r_2: `registro`) $\rightarrow$ res : `registro`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ copiarCampos(campos(r_2), r_1, r_2)}
Complejidad: $O(L)$, donde L es el dato string más largo de r_1.
Descripción: devuelve la unión de dos registros, pero sin campos repetidos.

---

iMerge(**in** r_1: `registro`, **in** r_2: `registro`) $\rightarrow$ res : registro

$res \leftarrow Copiar(r_1)$ /* L = dato string más largo de r1 */ ▷ $O\big(\#campos * (max\ nombre\ de\ campo\ +\ L)\big) = O(L)$
$ite \leftarrow VistaDicc(r_2)$ ▷ $O(1)$
while $HaySiguiente(ite)$ **do** ▷ $O(\#campos * ...) = O(1 * ...)$
  if not $Def?\big(Siguiente(it).clave, res\big)$ **then** ▷ $O(max\ nombre\ de\ campo) = O(1)$
    $Definir\big(Siguiente(it).clave, Siguiente(it).significado, res\big)$ ▷ $O(max\ nombre\ de\ campo\ *\ L) = O(L)$
  end if
  $Avanzar(it)$ ▷ $O(1)$
end while

Complejidad: $O(L)$, donde L es el dato string más largo de r_1.
Justificación:

Copiar el registro cuesta copiar #campos veces la clave y significados más costosos por interfaz de diccString. Como la cantidad de campos y los nombres de los campos están acotados, eso equivale a $O(L)$ siendo L el dato string más largo de r_1, y por lo tanto el más costoso de copiar. Por los mismos motivos se itera una cantidad acotada de veces; y preguntar si un campo está definido en un registro es también $O(1)$.

La inserción de cada <campo, dato>nuevo cuesta $O(max\ nombre\ de\ campo * L)$ (acotando) pero, nuevamente, los nombres de los campos están acotados y eso equivale al orden del dato más costoso.

---

iVistaJoin(**in** s_1 : `string`, **in** s_2 : `string`, **in** b: `estr`) → res : itConj(registro)

// convertimos s1 a tabla y preguntamos de qué tipo es su campoJoin con s2
$esNat \leftarrow TipoCampo?\Big(Obtener\big(s_2, \big(Obtener(s_1.hayJoin)\big)\big), Obtener(s_1, b.nombreATabla)\Big)$ ▷ $O(1)$

// campito = CAMPOJOIN
$campito \leftarrow Obtener\big(s_2, \big(Obtener(s_1.hayJoin)\big)\big).campoJoin$

// Creamos un iterador a la lista de cambios de tipo <registro, bool>de s1
$iteChanges \leftarrow CrearIt\Big(obtener\big(s_2, \big(obtener(s_1, b.hayJoin)\big)\big).cambios\Big)$ ▷ $O(1)$

if $esNat$ **then**
  // Join por campo nat
  $diccDeIters1 \leftarrow Obtener\big(s_1, Obtener\big(s_2, b.joinPorCampoNat\big)\big)$ ▷ $O(1)$
  $diccDeIters2 \leftarrow Obtener\big(s_2, Obtener\big(s_1, b.joinPorCampoNat\big)\big)$ ▷ $O(1)$

  // Guardo o elimino los registros en el join
  // Si no hay ninguno ⇒ No actualizo nada ⇒ O(1)
  while $HaySiguiente?(iteChanges)$ **do** /* R regs. en '.cambios' de s1 y s2 */ ▷ $O(R * ...)$
    $tupSiguiente \leftarrow Siguiente(itChanges)$ ▷ $O(1)$
    $claveNat \leftarrow Obtener(campito, tupSiguiente.reg)$ ▷ $O(1)$

    // Si no existe reg en s2 que tenga el mismo valor claveNat para 'campito', no necesito ni borrar ni agregar en el join
    $coincidencias \leftarrow Buscar(campito, claveNat, nombreATabla(s_2))$
      ▷ *Campo indexado* ⇒ $O(log\ m + |L|)$ / *Campo no indexado* ⇒ $O(m * |L|)$
    if $\#coincidencias > 0$ **then** ▷ $O(1)$
      // como campoJoin siempre es clave, #coincidencias es 1
      // regTablaActual es el registro en s2 que coincide en 'campito' con claveNat
      $regTablaActual \leftarrow Primero(coincidencias)$ ▷ $O(1)$
      if $tupSiguiente.agregar?$ **then** ▷ $O(1)$
        $registroMergeado \leftarrow Merge(tupSiguiente.reg, regTablaActual)$ ▷ $O(L)$
        $iter1 \leftarrow AgregarRapido\big(registroMergeado, Obtener(s_2, Obtener(s_1, e.registrosDelJoin))\big)$
          ▷ $O\big(copy(reg)\big) = O(L)$
        $iter2 \leftarrow AgregarRapido\big(registroMergeado, Obtener(s_1, Obtener(s_2, e.registrosDelJoin))\big)$ ▷ $O(L)$
        $Definir(claveNat, iter1, diccDeIters1)$
          /* m regs en s2, n regs en s1 */ ▷ $O\big(log\ (n+m)\big)\ +\ O\big(copy(iter)\big) = O\big(log(n+m)\big)$
        $Definir(claveNat, iter2, diccDeIters2)$ ▷ $O\big(log(n+m)\big)$
      else
        $EliminarSiguiente\big(Obtener(claveNat, diccDeIters1)\big)$ ▷ $O\big(log(n+m)\big)$
        $EliminarSiguiente\big(Obtener(claveNat, diccDeIters2)\big)$ ▷ $O\big(log(n+m)\big)$
        $Borrar(claveNat, diccDeIters1)$ ▷ $O\big(log(n+m)\big)$
        $Borrar(claveNat, diccDeIters2)$ ▷ $O\big(log(n+m)\big)$
      end if
    end if
    $EliminarSiguiente(iteChanges)$ ▷ $O(1)$
  end while

  // Ahora hago al reves, me fijo de la tabla 2 a la tabla 1
  $iteChanges \leftarrow CrearIt\Big(obtener\big(s_1, \big(obtener(s_2, b.hayJoin)\big)\big).cambios\Big)$ ▷ $O(1)$
  while $HaySiguiente?(iteChanges)$ **do** /* R regs. en '.cambios' de s1 y s2 */ ▷ $O(R * ...)$
    $tupSiguiente \leftarrow Siguiente(itChanges)$ ▷ $O(1)$
    $claveNat \leftarrow Obtener(campito, tupSiguiente.reg)$ ▷ $O(1)$

```
        // Pregunto si esta definido para no agregar el registro dos veces, solo pregunto en un dicc porque si esta definido en uno esta definido en el otro
        coincidencias ← Buscar(campito, claveNat, nombreATabla(s1))
                                    ▷ Campo indexado ⇒ O(log n + |L|) promedio / Campo no indexado ⇒ O(n * |L|)
        if #coincidencias > 0 and not Def?(claveNat, diccIters1) then                    ▷ O(log(n + m))
            // Como campoJoin siempre es clave, #coincidencias es 1
            regTablaActual ← Primero(coincidencias)                                      ▷ O(1)
            if tupSiguiente.agregar? then                                                ▷ O(1)
                registroMergeado ← Merge(tupSiguiente.reg, regTablaActual)               ▷ O(L)
                iter1 ← AgregarRapido(registroMergeado, Obtener(s2, Obtener(s1, e.registrosDelJoin)))
                                                                  ▷ O(copy(reg)) = O(L)
                iter2 ← AgregarRapido(registroMergeado, Obtener(s1, Obtener(s2, e.registrosDelJoin)))   ▷ O(L)
                Definir(claveNat, iter1, diccDeIters1)
                                                                  ▷ O(log(n + m))
                Definir(claveNat, iter2, diccDeIters2)                               ▷ O(log(n + m))
            else
                EliminarSiguiente(Obtener(claveNat, diccDeIters1))                   ▷ O(log(n + m))
                EliminarSiguiente(Obtener(claveNat, diccDeIters2))                   ▷ O(log(n + m))
                Borrar(claveNat, diccDeIters1)                                       ▷ O(log(n + m))
                Borrar(claveNat, diccDeIters2)                                       ▷ O(log(n + m))
            end if
        end if
        EliminarSiguiente(iteChanges)                                                ▷ O(1)
    end while
    res ← CrearIt(Obtener(s1, Obtener(s2, e.registrosDelJoin)))                      ▷ O(1)
else
    // Join por campo string
    diccDeIters1 ← Obtener(s1, Obtener(s2, b.joinPorCampoString))                    ▷ O(1)
    diccDeIters2 ← Obtener(s2, Obtener(s1, b.joinPorCampoString))                    ▷ O(1)

    while HaySiguiente?(iteChanges) do                                               ▷ O(R * ...)
        tupSiguiente ← Siguiente(itChanges)                                          ▷ O(1)
        claveString ← Obtener(campito, tupSiguiente.reg)                             ▷ O(1)
        coincidencias ← Buscar(campito, claveString, nombreATabla(s2))
                                    ▷ Campo indexado ⇒ O(L|) / Campo no indexado ⇒ O(m * |L|)
        if #coincidencias > 0 then                                                   ▷ O(1)
            // como campoJoin siempre es clave, #coincidencias es 1
            regTablaActual ← Primero(coincidencias)                                  ▷ O(1)
            if tupSiguiente.agregar? then                                            ▷ O(1)
                registroMergeado ← Merge(tupSiguiente.reg, regTablaActual)           ▷ O(L)
                iter1 ← AgregarRapido(registroMergeado, Obtener(s2, Obtener(s1, e.registrosDelJoin)))
                                                                  ▷ O(copy(reg)) = O(L)
                iter2 ← AgregarRapido(registroMergeado, Obtener(s1, Obtener(s2, e.registrosDelJoin)))   ▷ O(L)
                Definir(claveString, iter1, diccDeIters1)
                                                  ▷ O(L) + O(copy(iter)) = O(L)
                Definir(claveString, iter2, diccDeIters2)                            ▷ O(L)
            else
                EliminarSiguiente(Obtener(claveString, diccDeIters1))                ▷ O(L)
                EliminarSiguiente(Obtener(claveString, diccDeIters2))                ▷ O(L)
                Borrar(claveString, diccDeIters1)                                    ▷ O(L)
                Borrar(claveString, diccDeIters2)                                    ▷ O(L)
            end if
        end if
        EliminarSiguiente(iteChanges)                                                ▷ O(1)
    end while
```

```
    // Ahora hago al reves, me fijo de la tabla 2 a la tabla 1
    listCambios ← obtener(s1, (obtener(s2, b.hayJoin))).cambios                    ▷ O(1)
    iteChanges ← CrearIt(listCambios)                                              ▷ O(1)
    while HaySiguiente?(iteChanges) do                                             ▷ O(R * ...)
        tupSiguiente ← Siguiente(itChanges)                                        ▷ O(1)
        claveString ← Obtener(campito, tupSiguiente.reg)                           ▷ O(1)
        coincidencias ← Buscar(campito, claveString, nombreATabla(s1))
                                          ▷ Campo indexado ⇒ O(|L|) / Campo no indexado ⇒ O(n * |L|)
        if #coincidencias > 0 and not Def?(claveString, diccIters1) then           ▷ O(|L|)
            // Como campoJoin siempre es clave, #coincidencias es 1
            regTablaActual ← Primero(coincidencias)                                ▷ O(1)
            if tupSiguiente.agregar? then                                          ▷ O(1)
                registroMergeado ← Merge(tupSiguiente.reg, regTablaActual)         ▷ O(L)
                iter1 ← AgregarRapido(registroMergeado, Obtener(s2, Obtener(s1, e.registrosDelJoin)))
                                                                  ▷ O(copy(reg)) = O(L)
                iter2 ← AgregarRapido(registroMergeado, Obtener(s1, Obtener(s2, e.registrosDelJoin)))   ▷ O(L)
                Definir(claveString, iter1, diccDeIters1)
                                                  ▷ O(L) + O(copy(iter)) = O(L)
                Definir(claveString, iter2, diccDeIters2)                          ▷ O(L)
            else
                EliminarSiguiente(Obtener(claveString, diccDeIters1))              ▷ O(L)
                EliminarSiguiente(Obtener(claveString, diccDeIters2))              ▷ O(L)
                Borrar(claveString, diccDeIters1)                                  ▷ O(L)
                Borrar(claveString, diccDeIters2)                                  ▷ O(L)
            end if
        end if
        EliminarSiguiente(iteChanges)                                              ▷ O(1)
    end while
    res ← CrearIt(Obtener(s1, Obtener(s2, e.registrosDelJoin)))                    ▷ O(1)
end if
```

Complejidad:

Campo nat indexado ⇒

$O(R)$ * $\big(O(logm + |L|) + O(|L|) + O\big(log(n+m)\big)\big) + O(R)$ * $\Big(O(logn + |L|) + O(|L|) + O\big(log(n+m)\big)\Big) =$
$O(R)$ * ($O(log\ m + |L|) + O(|L|) + O\big(log(n+m)\big) + O(log\ n + |L|) + O(|L|) + O\big(log(n+m)\big)$) =
$O(R)$ * $\big(O(|L|) + O\big(log(n+m)\big) + O(logm) + O(logn)\big) =$
$O\big(R * \big(|L| + log(n+m) + log(m) + log\ n\big)\big) = O\big(R * \big(|L| + log(n+m) + log(n*m)\big)\big) =$
$O\big(R * \big(|L| + log(n*m)\big)\big)$

Campo nat no indexado ⇒
$O(R)$ * $\big(O(m * |L|) + O(|L|) + O\big(log(n+m)\big)\big) + O(R)$ * $\big(O(n * |L|) + O(|L|) + O\big(log(n+m)\big)\big) =$
$O(R)$ * $\big(O(m * |L|) + O(|L|) + O\big(log(n+m)\big) + O(n * |L|) + O(|L|) + O\big(log(n+m)\big)\big) =$
$O\big(R * (m * |L| + |L|) + log(n+m) + n * |L| + |L| + log(n+m)\big) =$
$O\big(R * \big(|L| * (m+n+2) + log(n+m) + log(n+m)\big)\big) =$
$O\big(R * \big(|L| * (m+n) + log(n+m)\big)\big)$

Campo string indexado ⇒
$O(R)$ * $O(|L|) + O(R)$ * $O(|L|) = O(R * |L|)$

Campo string no indexado ⇒ $O(R)$ * $\big(O(m * |L|) + O(|L|)\big) + O(R)$ * $\big(O(n * |L|) + O(|L|)\big) = O\big(R * (m * |L| + |L| + n * |L| + |L|)\big) = O\big(R * (|L|(n+m+2))\big) = O\big(R * |L| * (n+m)\big)$

L, cota para toda longitud de dato string en las dos tablas
n y m, cantidad de registros de las tablas con nombre s1 y s2 respectivamente
R, cantidad de registros a 'actualizar' (unión de las listas de cambios de ambas tablas)

Justificación:

Por cada uno de los R registros a actualizar se determina si se borran o se agregan. En peor caso se agregan (para borrar solo hace falta eliminar el siguiente de cada iterador y luego borrar la clave del diccionario): buscan coindencias en la otra tabla (complejidad varía según caso str/nat, indexado/no indexado), si las hay se debe hacer el merge en $O(L)$ e insertar al conjunto de registros $\big(O(|L|)$ para agregar registros por copia, por tener nombres y cantdidad de campos acotados, solo se paga por su máximo valor string una cantidad acotada de veces$\big)$. Finalmente se agrega el iterador a ese conjunto, también copiado en $O(1)$, a su respectivo diccionario de iteradores según tipo de campo $\big(O(|L|)$ para campos string, $O(log(n+m))$ para campos nat, dado que en el peor caso todos los registros de ambas tablas están en el join$\big)$.

Se repite el proceso para los elementos de la otra tabla, pero agregando el costo de preguntar si ya fueron definidos en el procedimiento anterior $\big($también $O(|L|)$ para campos string y $O(log(n+m))$ para campos nat$\big)$

iTablaMaxima(**in** b: `estr`) $\rightarrow$ res : string

$res \leftarrow *(b.tablaMasAccedida)$ ▷ $O(1)$

Complejidad: $O(1)$

Justificación: El algoritmo pasa por referencia un string, por lo tanto es $O(1)$.