DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

27 de mayo de 2014 — Algoritmos y Estructura de Datos II

Grupo 13

Integrante	LU	Correo electrónico
Fosco, Martin Esteban	449/13	mfosco2005@yahoo.com.ar
Minces Müller, Javier Nicolás	231/13	javijavi1994@gmail.com
Murga, Christian Mariano	982/12	christianmmurga@gmail.com
Palladino, Julian Alberto	336/13	julianpalladino@hotmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

1. Módulo Promesa

Interfaz

se explica con: PROMESA.

géneros: `promesa`.

Operaciones

TÍTULO(**in** p: `promesa`) $\rightarrow$ res : `nombre`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} ttulo(p)\}$
Complejidad: $O(1)$
Descripción: Devuelve el nombre del título sobre el que se hizo la promesa
Aliasing: res es modificable si y solo si p es modificable

TIPO(**in** p: `promesa`) $\rightarrow$ res : `tipoPromesa`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} tipo(p)\}$
Complejidad: $O(1)$
Descripción: Devuelve 'comprar' si la promesa es de compra y 'vender' si no lo es
Aliasing: res es modificable si y solo si p es modificable

LÍMITE(**in** p: `promesa`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} lmite(p)\}$
Complejidad: $O(1)$
Descripción: Devuelve el valor que la cotización tiene que superar para que se ejecute una promesa.

CANTIDAD(**in** p: `promesa`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} cantidad(p)\}$
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de acciones que se compran o se venden cuando la promesa se ejecuta

CREARPROMESA(**in** n: `nombre`, **in** t: `tipo`, **in** l: `dinero`, **in** c: `nat`) $\rightarrow$ res : `promesa`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} crearpromesa(n, t, l, c)\}$
Complejidad: $O(1)$
Descripción: Si t es 'comprar', devuelve una promesa de comprar c acciones de n si su coitzación supera l. Si t es 'vender', devuelve una promesa de vender c acciones de n si su cotización daja de l
Aliasing: n y t son modificables si y solo si la promesa es modificable

PROMESAEJECUTABLE(**in** p: `promesa`, **in** d: `dinero`, **in** n: `nat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} promesaEjecutable(p, d, n)\}$
Complejidad: $O(1)$
Descripción: Si la promesa es de compra, devuelve true si y solo si la cotización del título de la promesa superó el límite y hay suficientes disponibles. Si la promesa es de venta, devuelve true si y solo si la cotización del título bajó del límite.

COMPRAVENTA(**in** d: `nat`, **in** ps: `conj(promesa)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} compraVenta(d, ps)\}$
Complejidad: $O(\#(ps))$
Descripción: Devuelve la disponiblidad más la suma de cantidades de las promesas de compra de un conjunto, menos la suma de las cantidades de las promesas de venta de dicho conjunto.

promesasSobreTítulo(**in** n: `nombre`, **in** t: `tipoPromesa`, **in** ps: `conj(promesa)`) $\rightarrow$ res : `conj(promesa)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} promesasSobreTtulo(n, t, ps)$}
Complejidad: $O(\#(ps))$
Descripción: Devuelve, de un conjunto de promesas, aquellas que son de un determinado tipo y título.
Aliasing: Las promesas se devuelven por copia

Representación

`promesa` **se representa con** `estr`

donde `promesa` es `tupla`(*título*: `nombre` , *tipo*: `tipoPromesa` , *cantidad*: `acciones` , *límite*: `nat`)

Rep : estr $\longrightarrow$ bool
Rep(p) $\equiv$ true $\Longleftrightarrow$ true

Abs:estr p $\rightarrow$ promesa { Rep(p) }
Abs(p) $=_{\text{obs}}$ p2:promesa|título(p2)=p.título
$\wedge$ tipo(p2)=p.tipo
$\wedge$ cantidad(p2)=p.cantidad
$\wedge$ límite(p2)=p.límite

Algoritmos

(Las complejidades junto a una asignacion representan el costo de evaluar y asignar. Las complejidades junto a un **if** o un **while** representan el costo de evaluar la guarda. Las complejidades junto a un **endwhile** representan el costo de todo el ciclo.)

iTítulo(**in** p: `estr`) $\rightarrow$ res:nombre
    res $\leftarrow$ p.título — O(1)
iTipo(**in** p: `estr`) $\rightarrow$ res:tipoPromesa
    res $\leftarrow$ p.tipo — O(1)
iLímite(**in** p: `estr`) $\rightarrow$ res:dinero
    res $\leftarrow$ p.límite — O(1)
iCantidad(**in** p: `estr`) $\rightarrow$ res:nat
    res $\leftarrow$ p.cantidad — O(1)
iCrearPromesa(**in** n: `nombre`, **in** t: `tipo`, **in** l: `dinero`, **in** c: `nat`) $\rightarrow$ res:estr
    res.nombre $\leftarrow$ n — O(1)
    res.tipo $\leftarrow$ t — O(1)
    res.límite $\leftarrow$ l — O(1)
    res.cantidad $\leftarrow$ c — O(1)
iPromesaEjecutable(**in** p: `promesa`, **in** d: `dinero`, **in** n: `nat`) $\rightarrow$ res:bool
    res $\leftarrow$ tipo(p)=venta $\wedge$ d <límite(p)) $\vee$ tipo(p) = compra $\wedge$ d >límite(p) $\wedge$ n $\geq$ cantidad(p) — O(1)
iCompraVenta(**in** d: `nat`, **in** p: `conj(promesa)`) $\rightarrow$ res:nat
    res $\leftarrow$ d — O(1)
    ps $\leftarrow$ crearIt(p) — O(1)
    while (hayMas?(ps)) — O(1)
        if siguiente(ps).tipo=comprar — O(1)
            res $\leftarrow$ res+siguiente(ps).cantidad — O(1)
        else
            res $\leftarrow$ res-siguiente(ps).cantidad — O(1)
        endif
        avanzar(ps) — O(1)
    endwhile — O(#(p))
iPromesasSobreTítulo(**in** n: `nombre`, **in** t: `tipoPromesa`, **in** p: `conj(promesa)`) $\rightarrow$ res:conj(promesa)
    res $\leftarrow$ Vacio() — O(1)
    ps $\leftarrow$ crearIt(p) — O(1)
    while (hayMas?(ps)) — O(1)
        if siguiente(ps).tipo=t $\wedge$ siguiente(ps).título=n — O(1)
            AgregarRapido(res, siguiente(ps)) — O(1)

endif
avanzar(ps) O(1)
endwhile O(#(p))

2. Módulo Título

Interfaz

se explica con: TÍTULO.

géneros: `título`.

Operaciones

NOMBRE(**in** t: `título`) $\rightarrow$ res : `nombre`
Pre $\equiv$ {true}
Post $\equiv$ {$\widehat{res} =_{\text{obs}} nombre(\widehat{t})$}
Complejidad: $O(1)$
Descripción: Devuelve el nombre del título.
Aliasing: res es modificable si y solo si t es modificable.

CANTMAX(**in** t: `título`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$\widehat{res} =_{\text{obs}}$ #máxAcciones(t)}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad máxima de acciones disponibles para wolfie.

COT(**in** t: `título`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$\widehat{res} =_{\text{obs}}$ cotización($\widehat{t}$)}
Complejidad: $O(1)$
Descripción: Devuelve la cotización actual del título.

ENALZA(**in** t: `título`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$\widehat{res} =_{\text{obs}}$ enAlza($\widehat{t}$)}
Complejidad: $O(1)$
Descripción: Devuelve true si la cotización del título al recotizar es mayor a la anterior, o si nunca se aplicó recotizar sobre ese título, y false en caso contrario.

NUEVOT(**in** nt: `nombre`, **in** cot: `dinero`, **in** $cantMax$: `nat`) $\rightarrow$ res : `título`
Pre $\equiv$ {true}
Post $\equiv$ {nombre($\widehat{res}$) $=_{\text{obs}}$ $\widehat{nt}$ $\wedge$ #máxAcciones($\widehat{res}$) $=_{\text{obs}}$ $\widehat{cantMax}$ $\wedge$ cotización($\widehat{res}$) $=_{\text{obs}}$ $\widehat{cot}$ $\wedge$ enAlza($\widehat{res}$) $=_{\text{obs}}$ true}
Complejidad: $O(1)$
Descripción: Crea un nuevo título cuyos nombre, cotización y cantidad de acciones disponibles son aquellos pasados por parámetro
Aliasing: res es modificable si y solo si nt es modificable

RECOTIZAR(**in** cot: `dinero`, **in/out** t: `título`)
Pre $\equiv$ {$\widehat{t} =_{\text{obs}} t_0$}
Post $\equiv$ {$\widehat{t} =_{\text{obs}}$ recotizar($\widehat{cot}$, t_0)}
Complejidad: $O(1)$
Descripción: Cambia la cotización del título por la cot que se indique

cotActual(**in** *nt*: `nombre`, **in** *ts*: `conj(título)`) → *res* : `nat`
Pre ≡ {(∃ t:título)(t ∈ $\widehat{ts}$ ∧ nombre(t) =$_{obs}$ $\widehat{nt}$)}
Post ≡ {(∃ t:título)(t ∈ $\widehat{ts}$ ∧ nombre(t) =$_{obs}$ $\widehat{nt}$ ∧ cotización(t) =$_{obs}$ $\widehat{res}$)}
Complejidad: $O(Cardinal(ts))$
Descripción: Devuelve la cotización del título cuyo nombre coincide con el pasado por parámetro

cambiarCotTs(**in** *nt*: `nombre`, **in** *cot*: `dinero`, **in/out** *ts*: `conj(título)`)
Pre ≡ {(∃ t:título)(t ∈ $\widehat{ts}$ ∧ nombre(t) =$_{obs}$ $\widehat{nt}$) ∧ $\widehat{ts}$ =$_{obs}$ ts_0}
Post ≡ {$\widehat{ts}$ =$_{obs}$ cambiarCotización($\widehat{nt}$, $\widehat{cot}$, ts_0)}
Complejidad: $O(Cardinal(ts))$
Descripción: Cambia la cotización del título cuyo nombre coincide con el pasado por parámetro por la cotización (también pasada por parámetro)

limTen(**in** *nt*: `nombre`, **in** *ts*: `conj(título)`) → *res* : `nat`
Pre ≡ {(∃ t:título)(t ∈ $\widehat{ts}$ ∧ nombre(t) =$_{obs}$ $\widehat{nt}$)}
Post ≡ {(∃ t:título)(t ∈ $\widehat{ts}$ ∧ nombre(t) =$_{obs}$ $\widehat{nt}$ ∧ #máxAcciones(t) =$_{obs}$ $\widehat{res}$)}
Complejidad: $O(Cardinal(ts))$
Descripción: Devuelve la cantidad máxima de acciones disponibles para wolfie del título cuyo nombre coincide con el pasado por parámetro

titEnAlza(**in** *nt*: `nombre`, **in** *ts*: `conj(título)`) → *res* : `bool`
Pre ≡ {(∃ t:título)(t ∈ $\widehat{ts}$ ∧ nombre(t) =$_{obs}$ $\widehat{nt}$)}
Post ≡ {(∃ t:título)(t ∈ $\widehat{ts}$ ∧ nombre(t) =$_{obs}$ $\widehat{nt}$ ∧ enAlza(t) =$_{obs}$ $\widehat{res}$)}
Complejidad: $O(Cardinal(ts))$
Descripción: Informa si las acciones del título cuyo nombre coincide con el pasado por parámetro están en alza

Representación

Representación

donde `título` es `tupla`(*nombre*: `string` , *cantMax*: `nat` , *cot*: `nat` , *enAlza*: `bool`)

Rep : estr ⟶ bool
Rep(*t*) ≡ true ⟺ true

Abs:estr t → título { Rep(t) }
Abs(t) =$_{obs}$ t2:título|nombre(t2)=t.nombre,
∧#maxAcciones(t2)=t.cantMax
∧ cotizacion(t2)=t.cot
∧ enAlza(t2)=t.enAlza

Algoritmos

inombre(**in** *t*: `título`)→ res:string
res ← t.nombre $O(1)$

icantMax(**in** *t*: `título`) → res:nat

res ← t.cantMax $O(1)$

icot(**in** *t*: `título`)→ res:nat
res ← t.cot $O(1)$

ienAlza(**in** *t*: `título`)→ res:bool
res ← t.enAlza $O(1)$

inuevoT(**in** *nt*: `nombre`, **in** *cantMax*: `nat`, **in** *cot*: `dinero`)→ res:estr

res.nombre ← nt $O(1)$
res.cantMax ← cantMax $O(1)$
res.cot ← cot $O(1)$
res.enAlza ← true $O(1)$

IRECOTIZAR(**in** *cot*: `dinero`, **in/out** *t*: `estr`)
t.enAlza ← cot>t.cot $O(1)$
t.cot ← cot $O(1)$

ICOTACTUAL(**in** *nt*: `nombre`, **in** *ts*: `conj(estr)`) → res:nat
it ← CrearIt(ts) $O(1)$
while(HaySiguiente(it))
if(nombre(Siguiente(ts)) = nt) $O(1)$
res ← Siguiente(ts).cot $O(1)$
endif
avanzar(it) $O(1)$
endwhile $O(Cardinal(ts))$

ICAMBIARCOTTS(**in** *nt*: `nombre`, **in** *cot*: `dinero`, **in/out** *ts*: `conj(estr)`)
it ← CrearIt(ts) $O(1)$
while(HaySiguiente(it))
if(nombre(Siguiente(ts)) = nt) $O(1)$
Siguiente(ts).enAlza ← Siguiente(ts).cot<cot $O(1)$
Siguiente(ts).cot ← cot $O(1)$
endif
avanzar(it) $O(1)$
endwhile $O(Cardinal(ts))$

ILIMTEN(**in** *nt*: `nombre`, **in** *ts*: `conj(estr)`)
it ← CrearIt(ts) $O(1)$
while(HaySiguiente(it)) $O(1)$
if(nombre(Siguiente(ts)) = nt) $O(1)$
res ← Siguiente(ts).cantMax $O(1)$
endif
avanzar(it) $O(1)$
endwhile $O(Cardinal(ts))$

ITITENALZA(**in** *nt*: `nombre`, **in** *ts*: `conj(estr)`)
it ← CrearIt(ts) $O(1)$
while(HaySiguiente(it)) $O(1)$
if(nombre(Siguiente(ts)) = nt) $O(1)$
res ← Siguiente(ts).enAlza $O(1)$
endif
avanzar(it) $O(1)$
endwhile $O(Cardinal(ts))$

3. Módulo Wolfie

Interfaz

se explica con: WOLFIE.

géneros: `wolfie`.

Operaciones

CLIENTES(**in** w: `wolfie`) $\rightarrow$ res : `itBi(cliente)`
Pre $\equiv$ {true}
Post $\equiv$ $\{\widehat{res} =_{\text{obs}} clientes(\widehat{w})\}$
Complejidad: O(1)
Descripción: Devuelve un iterador de los clientes de wolfie.
Aliasing: Al ser nats, se pasan por copia.

TÍTULOS(**in** w: `wolfie`) $\rightarrow$ res : `itUni(título)`
Pre $\equiv$ {true}
Post $\equiv$ $\{\widehat{res} =_{\text{obs}} Títulos(\widehat{w})\}$
Complejidad: O(1)
Descripción: Devuelve un iterador de los títulos de wolfie.
Aliasing: Los títulos se pasan por referencia. Res es modificable si los títulos son modificables.

PROMESASDE(**in** c: `cliente`, **in** w: `wolfie`) $\rightarrow$ res : `itBi(promesa)`
Pre $\equiv$ $\{\widehat{c} \in clientes(\widehat{w})\}$
Post $\equiv$ $\{res =_{\text{obs}} promesasDe(c, w)\}$
Complejidad: $O(T \cdot C \cdot |max_n t|)$ y O(1) para sucesivas llamadas con el mismo c, salvo que se use actualizarCotizacion antes
Descripción: Devuelve todas las promesas del cliente c
Aliasing: Las promesas se pasan por copia.

ACCIONESPORCLIENTE(**in** c: `cliente`, **in** nt: `nombre`, **in** w: `wolfie`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{\widehat{c} \in clientes(\widehat{w}) \wedge (\exists t$:título$)(t \in$ títulos$(\widehat{w}) \wedge nombre(t) = \widehat{nt})\}$
Post $\equiv$ $\{\widehat{res} =_{\text{obs}} accionesPorCliente(\widehat{c}, \widehat{nt}, \widehat{w})\}$
Complejidad: $O(log(C) + |nt|)$
Descripción: Devuelve las acciones que tiene el cliente c del título cuyo nombre es nt

INAUGURAR WOLFIE(**in** cs: `conj(cliente)`) $\rightarrow$ res : `wolfie`
Pre $\equiv$ $\{\neg\emptyset(\widehat{cs})\}$
Post $\equiv$ $\{\widehat{res} =_{\text{obs}} inaugurarWolfie(\widehat{cs})\}$
Complejidad: $O(\#cs^2)$
Descripción: Crea un wolfie cuyos clientes son los de cs, sin títulos ni promesas.
Aliasing: Los elementos de cs se agregan por copia.

AGREGARTÍTULO(**in** t: `títiulo`, **in/out** w: `wolfie`)
Pre $\equiv$ $\{\widehat{w} =_{\text{obs}} w_0 \wedge_{\text{L}} (\forall\ t2$:título$)(t \in$ títulos$(\widehat{w}) \Rightarrow nombre(\widehat{t}) \neq nombre(t2))\}$
Post $\equiv$ $\{w =_{\text{obs}} agregarTítulo(\widehat{t}, w_0)\}$
Complejidad: $O(|nombre(t)| + C)$
Descripción: Agrega el título t a los títulos de wolfie
Aliasing: El elemento t se agrega por copia.

ACTUALIZARCOTIZACIÓN(**in** nt: `nombre`, **in** cot: `nat`, **in/out** w: `wolfie`)
Pre $\equiv$ $\{\widehat{w} =_{\text{obs}} w_0 \wedge_{\text{L}} \exists(t : título)(t \in títulos(\widehat{w}) \wedge nombre(t) = \widehat{nt})\}$
Post $\equiv$ $\{\widehat{w} =_{\text{obs}} actualizarCotización(\widehat{nt}, \widehat{cot}, w_0)\}$
Complejidad: $O(|nt| + C \cdot log(C))$
Descripción: Cambia la cotización del título nt por cot. Actualiza EnAlza y ejecuta las promesas
Aliasing: El elemento nt se pasa por copia

AGREGARPROMESA(**in** c: `cliente`, **in** p: `promesa`, **in/out** w: `wolfie`)
Pre $\equiv$ $\{\widehat{w} =_{\text{obs}} w_0 \wedge_{\text{L}} (\exists t : título)(t \in títulos(\widehat{w}) \wedge nombre(t) = título(\widehat{p})) \wedge \widehat{c} \in clientes(\widehat{w}) \wedge_{\text{L}} (\forall p2 : promesa)(p2 \in promesasDe(\widehat{c}, \widehat{w}) \Rightarrow (título(\widehat{p}) \neq título(p2) \vee tipo(\widehat{p}) \neq tipo(p2))) \wedge (tipo(\widehat{p}) = vender \Rightarrow accionesPorCliente(\widehat{c}, título(\widehat{p}), \widehat{w})$ $cantidad(\widehat{p})))\}$
Post $\equiv$ $\{\widehat{w} =_{\text{obs}} agregarPromesa(\widehat{nt}, \widehat{cot}, w_0)\}$
Complejidad: $O(|ttulo(p)| + log(C))$
Descripción: Agrega la promesa p al cliente c

Aliasing: La promesa se pasa por copia

ENALZA(**in** nt : `nombre`, **in** w : `wolfie`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{\exists(t : título)(t \in títulos(\widehat{w}) \wedge nombre(t) = \widehat{nt})\}$
Post $\equiv$ $\{\widehat{res} =_{\text{obs}} enAlza(\widehat{nt}, \widehat{w})\}$
Complejidad: $O(|nt|)$
Descripción: Devuelve true si el título cuyo nombre es nt está en alza, es decir, o bien su cotización es mayor a la que tenía al ejecutar por última vez actualizarCotizacion sobre ese título, o bien nunca se actualizó su cotización

Representación

`wolfie` **se representa con** `estr`

donde `estr` es `tupla`(*clientes*: `conj(cliente)`
, *títulos*: `conj(string)`
, *datosTítulo*: `diccT(string,tupla<título:título, cantWolfie:acciones, diccCli:diccOrd(cliente, datosCliente)>)`
, *accTxCliente*: `diccOrd(cliente, acciones)`
, *promUlt*: `conj(tupla<cliente:cliente, promesas:conj(promesa)>)`)

donde `datosCliente` es `tupla`(*acc*: `acciones`
, *pCompra*: `conj(promesa)`
, *pVenta*: `conj(promesa)`)

Descripcion del Rep:

1) Los clientes de w.Clientes son los mismos que las claves de w.accTxCliente y que las claves de diccCli para todo título. Ademas son no vacios.

2) Los nombres de w.Títulos son los mismos que las claves de w.datosTítulo.

3) Para todo cliente de W, la suma de sus acciones de todos los títulos es igual a su significado en w.accTxCliente

4) La suma de las acciones de todos los clientes sobre un título es igual a cantWolfie de ese título

5) Wolfie no puede tener mas acciones de un título que la cantidad maxima de ese título.

6) En cada significado de datosCliente no puede existir una promesa de Venta con una cantidad mayor a las acciones del cliente sobre el título.

7) Para cada significado de datosCliente, pCompra y pVenta son conjuntos vacios o de longitud 1. En pCompra solo puede haber de compra y en pVenta, de venta. Asimismo, el título de las promesas deben corresponder con la clave. (Dividimos en 7.1 para venta y 7.2 para compra)

8) PromUlt es vacio o de longitud 1 y tiene las promesas de un cliente sobre todos los títulos.

9) Los nombres de datosTítulo corresponden con el nombre del título.

Rep : estr $\longrightarrow$ bool

Rep(w) $\equiv$ true $\Longleftrightarrow$ **1)**(∀ t:nombre) t ∈ claves(w.datosTítulo) ⇒ (∀ c:cliente) c ∈ clientes ⇔ def?(c, w.accTxCliente) ⇔ def? (c,obtener(t, w.datosTítulo))) ∧¬∅?(w.clientes)

2) ∧(∀ t:nombre) t ∈ w.títulos ⇔ t ∈ claves(w.datosTítulo)

3) ∧(∀ c:cliente) c ∈ w.clientes $\Rightarrow_{\text{L}}$ obtener(c, w.accTxCliente) = $\sum_{t \in w.ttulos}$ (obtener(c, (obtener(nombre(t), w.diccTítulo)).datosCliente).acc)

4) ∧(∀ t:nombre) (def?(t, w.datosTítulo)) $\Rightarrow_{\text{L}}$ obtener(t,w.datosTítulo).accWolfie = $\sum_{c \in clientes(c)}$ TitXCli(c,t,w).acc

5) ∧(∀ t:nombre) (t ∈ w.títulos) $\Rightarrow_{\text{L}}$ (cantMax(obtener(t,datosTítulo)).título) ≤ (obtener(t,w.datosTítulo).cantWolfie))

6) ∧(∀ c:cliente, t:nombre) (c ∈ w.clientes ∧ def?(t,w.datosTítulo) $\wedge_{\text{L}}$¬∅?(TitXCli(c,t,w)).pVenta) $\Rightarrow_{\text{L}}$ cantidad(dameUno(TitXCli(c,t,w).pVenta)) ≤ TitXCli(c,t,w).acc

7.1) ∧(∀ c:cliente, t:nombre) (c ∈ w.clientes ∧ def?(t,w.datosTítulo)) $\Rightarrow_{\text{L}}$ (¬∅?(TitXCli(c,t,w)).pVenta $\Rightarrow_{\text{L}}$ (#(TitXCli(c,t,w).pVenta)=1 ∧ tipo(dameUno(TitXCli(c,t,w).pVenta))=vender) ∧ (nombre(dameUno(TitXCli(c,t,w).pVenta))=t)
7.2)∧¬∅?(TitXCli(c,t,w)).pCompra $\Rightarrow_{\text{L}}$ (#(TitXCli(c,t,w).pCompra)=1 ∧ tipo(dameUno(TitXCli(c,t,w).pCompra))=comprar)) ∧ (nombre(dameUno(TitXCli(c,t,w).pCompra))=t)

8)∧ (∅?(w.promUlt) $\vee_{\text{L}}$ ((w.promUlt)=1 ∧ (w.promUlt).cliente ∈ w.clientes) $\wedge_{\text{L}}$(∀ p:promesa) p ∈ (dameUno(w.promUlt).promesas) ⇔ (∃ n:nombre) (¬∅? (TitXCli(dameUno(w.promUlt).cliente,n,w).pCompra) $\wedge_{\text{L}}$ dameUno(TitXCli(dameUno(w.promUlt).cliente,n,w).pCompra)=p ∨¬∅? (TitXCli(dameUno(w.promUlt).cliente,n,w).pVenta) $\wedge_{\text{L}}$ dameUno(TitXCli(dameUno(w.promUlt).cliente,n,w).pVenta)

9) ∧(∀ t:nombre) def?(t,w.datosTítulo) $\Rightarrow_{\text{L}}$ nombre(obtener(t,w.datosTítulo).título)=t

TitXCli (c:cliente, t:nombre, w:wolfie) → datosCliente
TitXCli(c,t,w) = obtener(c,obtener(t, w.datosTítulo).datosCliente)

Abs:estr w → wolfie { Rep(w) }
Abs(w)=$_{\text{obs}}$ w2:wolfie|clientes(w2)=w.clientes
∧ títulos(w2)=$\bigcup_{t \in w.ttulos}(obtener(t, datosTtulo).ttulo)$,
∧ (∀ c:cliente) c ∈ clientes(w2) $\Rightarrow_{\text{L}}$ promesasDe(w2,c)=($\bigcup_{t \in w.ttulos}$ TitXCli(c,t,w).pVenta) ∪ ($\bigcup_{t \in w.ttulos}$ TitXCli(c,t,w).pVenta)
∧ (∀ c:cliente, nt:nombre) (c ∈ clientes(w2) ∧(∃ t:título) (nombre(t)=nt ∧ t ∈ títulos(w2))) $\Rightarrow_{\text{L}}$ Accionesporcliente(c,nt,w2)=obtener(c,(obtener(nt,e.datosTítulo)).acc)

Algoritmos

iClientes(**in** w: `estr`) → res:itConj(cliente)
  res ← crearIt(w.clientes) O(1)

iTítulos(**in** w: `estr`) → res:itDiccT(títulos)
  res ← crearIt(w.datosTítulo) O(1)

PromesasDe(**in** c: `cliente`, **in/out** w: `estr`) → res:itConj(promesa)
  prom ← Vacio() O(1)
  if esVacio?(w.promUlt) $\vee_{\text{L}}$ siguiente(crearIt(w.promUlt)).cliente ≠ c O(1)
    t ← crearIt(w.títulos) O(1)
    while hayMas(t) O(1)
      datosCli ← significado(c, significado(siguiente(t), w.datosTítulo).diccCli) O(|nt|+log(C))
      if ¬ esVacio(datosCli.pCompra) O(1)

agregarRapido(prom, siguiente(crearIt(datosCli.pCompra))) O(1)
endif
if ¬ esVacio(datosCli.pVenta) O(1)
agregarRapido(prom, siguiente(crearIt(datosCli.pVenta))) O(1)
endif
avanzar(t) O(1)
endwhile O(T(log(C) + |max_nt|))
res ← crearIt(prom) O(1)
w.promUlt ← Vacio() O(1)
agregarRapido(w.promUlt, <c,prom>) O(1)
else
res ← crearIt(siguiente(crearIt(w.promUlt)).promesas) O(1)
endif

iAccionesPorCliente(**in** c: `cliente`, **in** nt: `nombre`, **in** w: `estr`) → (res:nat)
datoCli ← significado(nt, w.datosTítulo).diccCli O(|nt|)
res ← significado(c, datoCli).acc O(log(C))
complejidad total: O(|nt|+log(C))

iInaugurarWolfie(**in** cl: `conj(cliente)`) → (res:estr)
cs ← crearIt(cl) O(C · log(C))
while haySiguiente(cs) O(1)
agregarRapido(res.clientes, siguiente(cs)) O(1)
definir(res.accTxCliente, siguiente(cs),0) O(C)
avanzar(cs) O(1)
endwhile O(#$(cl)^2$)
res.títulos ← Vacio() O(1)
res.promUlt ← Vacio() O(1)
Complejidad total: O(#$(cl)^2$)

iAgregarTítulo(**in** t: `título`, **in/out** w: `estr`)
t2 ← crearTítulo(nombre(t),cot(t),cantMax(t)) O(1)
agregarRapido(w.títulos, nombre(t)) O(1)
definir(w.datosTítulo, nombre(t2), <t2, 0, CrearDiccAccC(w.accTxCliente, <0, Vacio(), Vacio()>)>) O(|nt|+C)
Complejidad Total: O(|nt|+C)

crearDiccAccC (**in** cs: `diccOrd(cliente, acciones)`, **in** t: `tupla<nat, conj(promesa), conj(promesa)>`) → res:diccOrd(cliente,tupla<nat, conj(promesa), conj(promesa)>) **pre** ≡ { true }
post ≡ {(∀ c:cliente) definido?(c, res) $\Rightarrow_{\text{L}}$ obtener(c, res)=t }
c ← crearIt(cs)
while haySiguiente(cs) O(1)
definirRapido(siguiente(cs),t) O(1)
avanzar(cs) O(1)
endwhile O(C)

iActualizarCotización(**in** nt: `string`, **in** cot: `nat`, **in/out** w: `estr`)
datos ← significado(nt, w.datosTítulo) O(|nt|)
(datos.título) ← recotizar(datos.título) O(1)
itDiccCli ← crearIt(datos.diccCli)
while hayMas?(itDiccCli) O(1)
dcli ← siguienteSignificado(itDiccCli) O(1)
c ← siguienteClave(itDiccCli) O(1)
if #(dcli.pVenta)>0 $\wedge_{\text{L}}$ (promesaEjecutable(siguiente(crearIt(dcli.pCompra))), cot, ((datos.título).cantMax)-(datos.accWolfie)) O(1)
significado(c, w.accTxCliente) ← significado(c, w.accTxCliente-cantidad(siguiente(crearIt(dcli.pVenta))) O(log(C))
dcli.acc ← dcli.acc-cantidad(siguiente(crearIt(dcli.pVenta))) O(1)

dcli.pVenta ← Vacio() O(1)
w.PromUlt ← Vacio() O(1)
endif
avanzar(itDiccCli) O(1)
endwhile O(C · log(C))
arreglo(cliente, acciones) cliO ← diccOrd2ArrOrd(w.accTxCliente) O(C · log(C))
nat i ← 0 O(1)
while i<longitud(CliO) O(1)
dcli ← significado(siguienteClave(cliO),datos.diccCli) O(log(C))
c ← siguienteClave(datos.diccCli) O(1)
if #(dcli.pCompra)>0 $\wedge_L$ (promesaEjecutable(siguiente(crearIt(dcli.pCompra))), cot, ((datos.título).cantMax)-(datos.accWolfie)) O(1)
significado(c, w.accTxCliente) ← significado(c, w.accTxCliente+cantidad(siguiente(crearIt(dcli.pCompra))) O(log(C))
dcli.acc ← dcli.acc+cantidad(siguiente(crearIt(dcli.Compra))) O(1)
dcli.pCompra ← Vacio() O(1)
w.PromUlt ← Vacio() O(1)
endif
i ← i+1 O(1)
endwhile O(C · log(C))
Complejidad Total: O(|nt|+C · log(C))

DICCORD2ARRORD(**in** d: `diccOrd(cliente, acciones)`) → res:arreglo(tupla<(cliente, acciones)>)
Pre ≡ { ¬ esVacio?(d) }
Post ≡ { (∀ i, j: nat) (i<longitud(res)) ∧ (j<longitud(res)) $\Rightarrow_L$ (Π_2(res[i])≤ Π_2(res[j]) }
i ← 0 O(1)
di ← crearIt(d) O(1)
arr ← crearArreglo(#claves(d)) O(1)
while i <#claves(d)) O(1)
arr[i]=<siguienteClave(di), siguienteSignificado(di)> O(1)
i ← i+1 O(1)
avanzar(di) O(1)
endwhile O(C)
res ← Mergesort(arr) O(C · log(C)), Mergesort es igual al Mergesort apunte de algoritmos basicos pero aplicada sobre un arreglo de tuplas en vez de naturales. Sin embargo, como los segundos elementos de la tupla son nat, comparar dos tuplas por el segundo elemento cuesta O(1).

MERGE(**in** b: `arreglo(tupla<cliente, acciones>)`, **in** c: `arreglo(tupla<cliente, acciones>)` → res:arreglo(tupla<cliente, acciones>)
nat ib ← 1;
nat ic ← 1
res ← CrearArreglo(tam(b) + tam(c))
nat i ← 1
while i ≤ tam(a)
if iB ≤ tam(b) ∧ (ic >tam(c) ∨Π_2 b[ib] <Π_2c[ic])
res[i] ← b[ib]
ib ← ib + 1
else
res[i] ← c[ic]
ic ← iC + 1
endif
i ← i+1
endwhile

MERGESORT (**in/out** a: `arreglo(tupla<cliente, acciones>)`)
if tam(a)>1

```
        nat m ← tam(a)/2
        arreglo(nat) b ← Copiar(Subarreglo(a, 1, m))
        arreglo(nat) c ← Copiar(Subarreglo(a, m+1; tam(a)))
        MergeSort(b)
        MergeSort(c)
        a ← Merge(b, c)
    endif
```

```
iAgregarPromesa(in p: promesa, in c: cliente, in/out w: estr)
    datos ← significado(c, (significado(título(p), w.datosTítulo)))          O(log(C)+|nt|)
    if (tipo(p)=vender)                                                      O(1)
        agregarRapido(datos.pVenta, p)                                       O(1)
    else
        agregarRapido(datos.pCompra, p)                                      O(1)
    endif
    if ¬ esVacio?(w.promUlt) ∧_L siguiente(crearIt(w.promUlt))).cliente=c    O(1)
        agregarRapido(siguiente(crearIt(w.promUlt))).promesas, p)            O(1)
    endif
Complejidad total: O(log(C)+|nt|)

iEnAlza(in nt: nombre,in w: estr) → (res:bool)
    res ← EnAlza(significado(nt, w.datosTítulo))                             O(|nt|)
```

4. Módulo Diccionario Ordenado(clave, significado)

Interfaz

parametros formales

generos κ, σ

funcion $\bullet{=}\bullet$(**in** $k1 : \kappa$, **in** $k2 : \kappa$) $\rightarrow$ res : `bool`
- **Pre** $\equiv$ {true}
- **Post** $\equiv$ $\{res =_{\text{obs}} (c1 = c2)\}$
- **Complejidad:** $\Theta(equal(c))$
- **Descripción:** funcion de igualdad de κ's

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

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

funcion $\bullet{<}\bullet$(**in** $c1 : \kappa$ **in** $c2 : \kappa$) $\rightarrow$ res : κ
- **Pre** $\equiv$ {true}
- **Post** $\equiv$ $\{res =_{\text{obs}} (c1 < c2)\}$
- **Complejidad:** $\Theta(equal(c))$
- **Descripción:** funcion de orden de κ's

se explica con: Diccionario(clave, significado), modificado de la siguiente forma:

Observadores agregados:

maxClaves: dicc(clave, significado) → nat

Generadores modificados:

vacio : nat → dicc(clave, significado)
definir : clave × significado × dicc(clave, significado) d → dicc(clave, significado) {#(claves(d)) }
Nuevos axiomas:
maxClaves(Vacio(n)) ≡ n
maxClaves(definir(c,s,d)) ≡ maxClaves(d)

géneros: `diccOrd`, `itDiccOrd`

El Módulo Diccionario Ordenado provee un diccionario con una interfaz similar al diccionario lineal, es decir que muchas de sus operaciones básicas tienen los mismos argumentos de entrada y salida y cumplen los mismos Pre y Post. Se diferencian en las complejidades de las operaciones. Tiene también un máximo de claves que puede tener, que se especifica al crear el diccionario. Su iterador es unidireccional.

VACIO(**in** n: `nat`) → res : `diccOrd(`κ`, `σ`)`
Pre ≡ {true}
Post ≡ {res $=_{obs}$ vacio}
Complejidad: O(1)
Descripción: Crea un diccionario vacio cuyo maximo de claves es n

DEFINIR(**in** c: κ, **in** s: σ, **in/out** d: `diccOrd(`κ`, `σ`)`) → res : `itDiccOrd(`κ`, `σ`)`
Pre ≡ {d $=_{obs}$ d_0 ∧¬ definido?(d, c) ∧ #claves(d) ≠ maxClaves(d)}
Post ≡ {d $=_{obs}$ definir(d_0; c; s) ∧ haySiguiente(res) $\wedge_L$ Siguiente(res) = <c; s>∧ esPermutacion(SecuSuby(res),d)}
Complejidad: O(#claves(d) · (copy(κ)+copy(σ)))
Descripción: Define la clave c con el significado s en el diccionario
Aliasing: Los elementos c y s se definen por copia. El iterador se invalida si y solo si se elimina el elemento siguiente del iterador sin utilizar la funcion EliminarSiguiente. Ademas, siguientes(res) podria cambiar completamente ante cualquier operacion que modifique el d sin utilizar las funciones del iterador

DEFINIRRAPIDO(**in** c: κ, **in** s: σ, **in/out** d: `diccOrd(`κ`, `σ`)`) → res : `itDiccOrd(`κ`, `σ`)`
Pre ≡ {d $=_{obs}$ d_0 ∧¬ definido?(d, k) ∧(∀ c2 ∈ claves(d)) c<c2 ∧ #claves(d) ≠ maxClaves(d)}
Post ≡ {d $=_{obs}$ definir(d_0; c; s) ∧ haySiguiente(res) $\wedge_L$ Siguiente(res) = <c; s>∧ esPermutacion(SecuSuby(res),d)}
Complejidad: O(copy(κ)+copy(σ))
Descripción: Define la clave c con el significado s en el diccionario, requiere que c sea mayor a todo el resto de las claves del diccionario
Aliasing: Los elementos c y s se definen por copia. El iterador se invalida si y solo si se elimina el elemento siguiente del iterador sin utilizar la funcion EliminarSiguiente. Ademas, siguientes(res) podria cambiar completamente ante cualquier operacion que modifique el d sin utilizar las funciones del iterador
MAXCLAVES(**in** d: `diccOrd(`κ`, `σ`)`) → res : `nat`
Pre ≡ {true}
Post ≡ {d $=_{obs}$ maxClaves(d)}
Complejidad: O(1)
Descripción: Devuelve el maximo de claves del diccionario

Las complejidades de las otras operaciones básicas son listadas a continuación:

DEFINIDO?(c, s, d): O(log(#claves(d)) · equal(κ))
SIGNIFICADO(c, d): O(log(#claves(d)) · equal(κ))
BORRAR(c, d): O(#claves(d) · (equal(κ)+copy(κ)+copy(σ)))
#CLAVES(d): O(1)
COPIAR(d): O(#claves(d) · (copy(κ)+copy(σ)))
D1=D2: O(#claves(d) · (copy(κ)+copy(σ)))

Las operaciones del iterador definidas son las siguientes, con su complejidad:

CREARIT(D): O(1)
HAYSIGUIENTE(D): O(1)
SIGUIENTE(D): O(1)
SIGUIENTECLAVE(D): O(1)
SIGUIENTESIGNIFICADO(D): O(1)

AVANZAR(D): O(1)

Representación

Representación del diccionario

`diccOrd(`κ`, `σ`)` **se representa con** `diccord`

donde `diccOrd` es tupla<elementos:array(tupla<κ, σ>), longitud:nat, lReal:nat>

Rep : $\widehat{estr}$ $\longrightarrow$ bool
Rep(d) $\equiv$ true $\Longleftrightarrow$ ($\forall$ i,j:nat) (i<d.lReal $\wedge$ j<d.lReal) $\Rightarrow_{\text{L}}$ (Π_1 d.elementos[i] $\neq$ Π_1 d.elementos[j] $\wedge$ i<j $\Leftrightarrow$ Π_1 d.elementos[i]<Π_1 d.elementos[j])
$\wedge$ d.lReal $\leq$ d.longitud
$\wedge$($\forall$i: nat)(0 $\leq$ i <d.longitud $\Rightarrow_{\text{L}}$ def?(v.elementos, i)

Abs : estr d $\rightarrow$ dicc(κ, σ) { Rep(d)}
Abs(d) $=_{\text{obs}}$ d2: dicc(κ, σ) | ($\forall$ c:κ) def?(c, d2)=($\exists$ s:σ, i:nat) d.elementos[i]=<c, s>) $\wedge_{\text{L}}$($\forall$ c:κ) obtener(c, d2)=Π_2 (d.elementos[buscarIndice(d, c, 0)])

(buscarIndice) (estr, κ, nat) $\rightarrow$ nat buscarIndice(e, c, i)= **if** Π_1 e[i]=c **then** i **else** buscarIndice(e, c, i+1) **fi**

Representación del iterador

`itDiccOrd(`κ`, `σ`)` **se representa con** `itDiccOrd`

donde `itDiccOrd` es tupla<elementos: array(<tupla(κ), (σ)>$*$, i:nat, long:nat>)

Rep : $\widehat{iter}$ $\longrightarrow$ bool
Rep(id) $\equiv$ true $\Longleftrightarrow$ ($\forall$ i, j:nat) (i<id.long $\wedge$ j<id.long) $\Rightarrow_{\text{L}}$ i<j $\leftrightarrow$ Π_1(id.elementos$*$)[i] $\leq$ Π_1(id.elementos$*$)[j]
$\wedge$ id.i <id.long

Abs : iter it $\rightarrow$ itUni(α) { Rep(it)}
Abs(it) $=_{\text{obs}}$ b: itUni(α) | Siguientes(b) = Abs(subArreglo(it.elementos$*$, it.i, it.long))

Algoritmos

iVACIO (n:nat) $\rightarrow$ res:estr
res.elementos $\leftarrow$ a[n] O(1)
res.lReal $\leftarrow$ 0 O(1)
res.longitud $\leftarrow$ n O(1)

iDEFINIR (**in/out** d: `diccOrd(`κ`, `σ`)`, **in** c: κ, **in** s: σ) $\rightarrow$ res:itDiccOrd(κ, σ)
i $\leftarrow$ 0 O(1)
while i<d.lReal O(1)
if Π_1(d.elementos[i]) >c O(1)
<a, b>$\leftarrow$ copy(d.elementos[i]) O(copy(κ)+copy(σ))
d.elementos[i] $\leftarrow$ copy(<c,s>) O(copy(κ)+copy(σ))
<c, s>$\leftarrow$ copy(<a, b>) O(copy(κ)+copy(σ))
endif
i $\leftarrow$ i+1 O(1)
endwhile O(d.lreal $\cdot$ (copy(κ)+copy(σ)))
d.elementos[i] $\leftarrow$ copy(<c, s>) O(copy(κ)+copy(σ))
d.lReal $\leftarrow$ d.lReal+1 O(1)
res $\leftarrow$ crearIt(d)

iDEFINIRRAPIDO (**in/out** d: `diccOrd(`κ`, `σ`)`, **in** c: κ, **in** s: σ) $\rightarrow$ res:itDiccOrd(κ, σ)
d.lReal $\leftarrow$ d.lReal+1 O(1)
d.elementos[lReal+1] $\leftarrow$ copy(<c, s>) O(copy(κ)+copy(σ))
res $\leftarrow$ crearIt(d)

iDEFINIDO? (**in** d: `diccOrd` **in** c: κ) $\rightarrow$ res:bool
i $\leftarrow$ (d.lReal)/2 O(1)
max $\leftarrow$ d.lReal O(1)

```
    min ← 0                                                                    O(1)
    res ← false                                                                O(1)
    while max ≠ min ∧ max ≠ min+1 ∧ res=false                                  O(1)
        if Π1 d.elementos[i]=c                                                 O(equal(κ))
            res ← true                                                         O(1)
        else if Π1 d.elementos[i]<c                                            O(equal(κ))
            min ← i                                                            O(1)
            i ← (i+max)/2                                                      O(1)
        else                                                                   O(1)
            max ← i                                                            O(1)
            i ← (i+min)/2                                                      O(1)
        endif
    endwhile                                                                   O(log(#claves(d)) · equal(κ))

iSignificado (in d: diccOrd in c: κ) → res:σ
    i ← (d.lReal)/2                                                            O(1)
    max ← d.lReal                                                              O(1)
    min ← 0                                                                    O(1)
    encontrado ← false                                                         O(1)
    while ¬ encontrado                                                         O(1)
        if Π1 d.elementos[i]=c                                                 O(equal(κ))
            res ← Π1 d.elementos[i]                                            O(equal(κ))
            encontrado ← true                                                  O(1)
        else if Π1 d.elementos[i]<c                                            O(equal(κ))
            min ← i                                                            O(1)
            i ← (i+max)/2                                                      O(1)
        else                                                                   O(1)
            max ← i                                                            O(1)
            i ← (i+min)/2                                                      O(1)
        endif
    endwhile                                                                   O(log(#claves(d)) · equal(κ))

iBorrar(in/out d: diccOrd, in c: κ)
    i ← 0                                                                      O(1)
    encontrado ← false                                                         O(1)
    while i<(d.lReal-1)                                                        O(1)
        if Π1(d.elementos[i]) = c ∧ encontrado=true                            O(equal(κ))
            d.elementos[i] ← copy(d.elementos[i+1])                            O(copy(κ)+copy(σ))
            encontrado ← true                                                  O(1)
        endif
        i ← i+1                                                                O(1)
    endwhile                                                                   O(d.lReal · (equal(κ)+copy(κ)+copy(κ)))
    d.lReal ← d.lReal-1                                                        O(1)

i#Claves (in d: diccOrd) → res:nat
    res ← d.lReal                                                              O(1)

•=• (in d1: diccOrd(κ, σ), in d2: diccOrd(κ, σ)) → res:bool
    i ← 0                                                                      O(1)
    res ← true                                                                 O(1)
    if (d1.lreal ≠ d2.lReal)                                                   O(1)
        res ← false                                                            O(1)
    else
        while i<d1.lReal                                                       O(1)
            if d1[i] = d2[i]                                                   O(equal(κ)+equal(σ))
                res ← false                                                    O(1)
            endif
            i ← i+1                                                            O(1)
```

endwhile O(d1.lReal · (equal(κ)+equal(σ)))
endif

iCrearIt (**in** d: `DiccOrd(`κ`, `σ`)`) → res:itDiccOrd(κ, σ)
res.i ← 0 O(1)
res.long ← d.lReal O(1)
res.elementos ← *(d.elementos) O(1)

iHaySiguiente (**in** d: `itDiccOrd(`κ`, `σ`)`) → res:bool
res ← d.i<d.long O(1)

isiguienteClave (**in** d: `itDiccOrd(`κ`, `σ`)`) → res:κ
res ← Π_1(*d.elementos) O(1)

isiguienteSignificado (**in** d: `itDiccOrd(`κ`, `σ`)`) → res:σ
res ← Π_2(*d.elementos) O(1)

iSiguiente (**in** d: `itDiccOrd(`κ`, `σ`)`) → res:tupla<κ, σ>
res ← * d.elementos O(1)

iAvanzar (**in/out** d: `itDiccOrd(`κ`, `σ`)`)
d.i ← (d.i)+1 O(1)
d.elementos ← (d.elementos)+1 O(1)

5. Módulo Diccionario Texto(string, tupla(α, β, γ))

Interfaz

se explica con: DiccionarioEterno.

géneros: `diccT(string, tupla(`α`, `β`, `γ`))`

El Módulo Diccionario Texto provee un diccionario con la misma interfaz que la del Diccionario Lineal, salvo borrar, #Claves, copiar y comparación(es decir que sus operaciones básicas tienen los mismos nombres, mismos argumentos de entrada y salida y cumplen los mismos Pre y Post, salvo borrar que no está disponible) y las operaciones tienen distintas complejidades.

Además hicimos que se explique con el TAD DiccionarioEterno que tiene los mismos generadores, observadores y otras Operaciones que las del TAD Diccionario(κ, σ) (con la misma axiomatización) exceptuando borrar, que no está en DiccionarioEterno. Consideramos que todos los significados de DiccionarioEterno son tupla(α, β, γ) y que las claves son strings.

el iterador devuelve elementos de tipo α al aplicar siguiente, no tuplas.

Las complejidades de las operaciones básicas son listadas a continuación:

Vacio(): O(1)
Definir(s:string, tupla(α, β, γ), d:diccT(string, tupla(α, β, γ))): O(long(s))
Definido?(s:string, d:diccT(string, tupla(α, β, γ))): O(long(s))
Significado(s:string, d:diccT(string, tupla(α, β, γ))): O(long(s))

Las complejidades de las operaciones del iterador son las siguientes:

CrearIt(d:diccT(string, tupla(α, β, γ))): O(1)
Avanzar(id:itdiccT(string, tupla(α, β, γ))): O(1)
SiguienteClave(id:itdiccT(string, tupla(α, β, γ))): O(1)
SiguienteSignificado(id:itdiccT(string, tupla(α, β, γ))): O(long(SiguienteClave(id)))
Siguiente(id:itdiccT(string, tupla(α, β, γ))): O(long(SiguienteClave(id)))

HaySiguiente(id:itdiccT(string, tupla(α, β, γ))): O(long(SiguienteClave(id)))

Representación

Representación del Diccionario Texto

DiccT se representa con dicct

tupla(primVec:vectorDe256Punteros(ptro(vectorDe256Punteros)),
claves:conj(string))

Rep : dicct $\longrightarrow$ bool
Rep(d) $\equiv$ true $\Longleftrightarrow$ **1)** ($\forall$ s:string) (pertenece?(s, claves(d))) $\leftrightarrow$ (definido?(s,d))
1) Todos los elementos de claves estÃ¡n definidos y viceversa. **2)** Los punteros no forman ciclos (Para todos los caminos posibles de punteros en ningÃºn momento se puede recorrer dos veces al mismo)
Abs:dicct d $\rightarrow$ diccionarioEterno { Rep(d) }
Abs(d)=$_{obs}$ d2:dicct|($\forall$ s: string) (Definido?(s,d)) $\leftrightarrow$ (Definido?(s,d2))
$\wedge$ ($\forall$ s:string) (Definido?(s,d)) $\Rightarrow_{L}$ (Significado(s,d) = Significado(s,d2))

Representación del iterador

ItDiccT se representa con itdicct

tupla(punt: $*$(dicct), itClaves:itConj(string))

Descripcion del Rep:

Rep : itdicct $\longrightarrow$ bool
Rep(d) $\equiv$ true $\Longleftrightarrow$ Rep($*$punt) $\wedge$ d.itClaves=claves($*$punt)

Abs:itdicct d $\rightarrow$ itUni(α) { Rep(d) }
Abs(d)=$_{obs}$ d2:itUni(α)| Siguientes(d2) = (significados($*$d.elementos, siguientes(crearIt(d.claves))))
significados (dicct, conj(claves)) $\rightarrow$ conj(α) significados(d, cs) = $\bigcup_{c \in cs} \Pi_1$(significado(c,d))

Algoritmos Vacio() $\rightarrow$ res:diccT(string, tupla(α, β, γ))

```
res ← tupla(<NULL,NULL,...,NULL>, Vacio())                                  O(1)
```

Definir(**in** s: `string`, **in** sig: `tupla(`α`, `β`, `γ`)`, **in/out** d: `diccT(string, tupla(`α`, `β`, `γ`))`)

```
i ← 1                                                                       O(1)
paux ← d.primVec[ord(s[0])]                                                 O(1)
while (i<Longitud(s))                                                       O(1)
    if (*paux[ord(s[0])]=NULL                                               O(1)
        nuevoVec ← tupla(<NULL, NULL,...,NULL>,NULL)
        (*paux)[ord(s[i])]=&nuevoVec
    endif
    paux.sign ← sig
    i = i + 1                                                               O(1)
endwhile                                                                    O(Longitud(s))
res ← (*pAux).sign                                                          O(1)
agregarRapido (d.claves, s)                                                 O(long(s))
```

Definido?(**in** s: `string` ,**in** d: `dicct(string, tupla(`α`, `β`, `γ`))`) $\leftarrow$ res:bool

```
res ← true                                                                  O(1)
if d.primVec[ord([s(0)])]=NULL                                              O(1)
    res ← false                                                             O(1)
endif                                                                       O(1)
i ← 1                                                                       O(1)
paux ← d.primVec[ord(s[0])]                                                 O(1)
while(i<Longitud(s))                                                        O(1)
paux ← paux[ord(s[i])]                                                      O(1)
```

if $*$ paux [ord(s[i])]=NULL — O(1)
res $\leftarrow$ false — O(1)
endif — O(1)
i = i + 1 — O(1)
endwhile — O(longitud(s))
res $\leftarrow$ ($*$pAux).sign — O(1)

SIGNIFICADO(**in** s: `string`, **in** d: `dicct(string, tupla(`α`, `β`, `γ`))`) $\leftarrow$ res:tupla(α, β, γ)
i $\leftarrow$ 1 — O(1)
paux $\leftarrow$ d.primVec[ord(s[0])] — O(1)
(i<longitud(s)) — O(1)
while i<longitud(s) — O(1)
paux $\leftarrow$ (paux.VecPunteros[ord(s[i])]) — O(1)
i $\leftarrow$ i+1 — O(1)
endwhile — O(longitud(s))
res $\leftarrow$ ($*$Paux).sign — O(1)

CREARIT (**in** d: `dicct(string, tupla(`α`, `β`, `γ`))`) $\rightarrow$ res: itDiccT(string, tupla(α, β, γ))
res.punt $\leftarrow$ & d — O(1)
res.itClaves $\leftarrow$ crearIt(d.claves) — O(1)

AVANZAR (**in**/**out** d: `dicct(string, tupla(`α`, `β`, `γ`))`)
avanzar (d.itClaves) — O(1)

SIGUIENTECLAVE (**in** d: `dicct(string, tupla(`α`, `β`, `γ`))`) $\rightarrow$ res:string
res $\leftarrow$ siguiente(d.itClaves) — O(1)

SIGUIENTESIGNIFICADO (**in** d: `dicct(string, tupla(`α`, `β`, `γ`))`) $\rightarrow$ res:tupla(α, β, γ))
res $\leftarrow$ significado(($*$(d.punt)),siguiente(res.claves)) — O(longitud(siguiente(res.claves)))

SIGUIENTE (**in** d: `dicct(string, tupla(`α`, `β`, `γ`))`) $\rightarrow$ res:α
res $\leftarrow$ Π_1(significado(($*$(d.punt)),siguiente(res.claves))) — O(longitud(siguiente(res.claves)))

HAYSIGUIENTE (**in** d: `dicct(string, tupla(`α`, `β`, `γ`))`) $\rightarrow$ res:bool
res $\leftarrow$ haySiguiente(d.claves) — O(1)

6. Informe

6.1. Decisiones Tomadas

1: Decidimos llamar al Diccionario Ordenado debido a que el iterador devuelve de manera ordenada sus tuplas, por lo tanto no hay problemas de encapsulamiento.

6.2. Soluciones alternativas

Para poder usar un tipo σ general en lugar de una tupla $<\alpha, \beta, \gamma>$ en el diccionario de texto, se podria modificar la estructura de forma que incluya dos de estos diccionarios: uno cuyo significado sean titulos y otro cuyo significado sea el resto de los datos necesarios. De esta forma, el iterador podría devolver elementos de tipo σ, no buscar el primer elemento de la tupla, y se seguirían cumpliendo las complejidades.
En la estructura preferimos usar conjuntos de hasta una promesa a usar un puntero a promesa que se haría NULL si no hubiera ninguna.