Algoritmos y Estructuras de Datos II

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

RTP 2 - Broker System

Grupo 20

Integrante	LU	Correo electrónico
Fernando Gasperi Jabalera	56/09	fgasperijabalera@gmail.com
Esteban Romero	659/06	estebantaborcias@gmail.com
Leandro Tozzi	-	leandro.tozzi@gmail.com
Alfredo Terrile Cendoya	022/11	freddy199_0@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Observaciones generales

Convenciones que adoptamos en todos los móduos:

- nos referimos a los campos de las tuplas por el nombre de los mismos, no por $\prod_1$, $\prod_2$,...,$\prod_n$.
- en los algoritmos utilizamos los alias de los tipos de tuplas que definimos en la estructura de representación. Por ejemplo, si en la estructura de representación definimos una tupla que la llamamos tuplaEspecial:
  donde `tuplaEspecial` es `tupla`(*campoEspecial*$_1$: `tipo`$_1$, *campoEspecial*$_2$: `tipo`$_2$, *campoEspecial*$_3$: `tipo`$_3$)
  después en los algoritmos cada vez que usemos una tupla con esos tipos usamos el alias tuplaEspecial y nos referimos a sus campos por campoEspecial$_1$, campoEspecial$_2$ y campoEspecial$_3$.

2. Módulo Wolfie

Interfaz

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

se explica con: WOLFIE.

géneros: `wolfie`.

Operaciones básicas de Wolfie

INAUGURARWOLFIE(**in** *clientes*: `conj(clientes)`) $\rightarrow$ *res* : `wolfie`
Pre $\equiv$ {$\neg\emptyset?(clientes)$}
Post $\equiv$ {$res =_{\text{obs}} inaugurarWolfie(clientes)$}
Complejidad: $O(\#(clientes)^2)$
Descripción: genera un wolfie con los clientes recibidos en *clientes*.

AGREGARTÍTULO(**in/out** w: `wolfie`, **in** *nomTit*: `string`, **in** *maxAcciones*: `nat`, **in** *cot*: `nat`)
Pre $\equiv$ {$w =_{\text{obs}} w_0 \wedge (\forall\ t : titulo)\ (t \in \text{títulos}(w_0) \Rightarrow \text{nombre}(t) \neq \text{nomTit})$ }
Post $\equiv$ {$w =_{\text{obs}}$ agregarTítulo(crearTítulo(*nomTit*, *cot*, *maxAcciones*), w_0)}
Complejidad: $\Theta(copy(a))$
Descripción: agrega un título a w con el nombre *nt*, la cotización *cotizacion* y un tope máximo de acciones *maxAcciones*.

ACTUALIZARCOTIZACIÓN(**in/out** w: `wolfie`, **in** *nomTit*: `string`, **in** *cot*: `nat`)
Pre $\equiv$ {$w =_{\text{obs}} w_0 \wedge (\exists$ *t:titulo*$)\ (t \in \text{títulos}(w_0) \wedge \text{nombre}(t) = \text{nomTit})$}
Post $\equiv$ {$w =_{\text{obs}}$ actualizarCotización(*nomTit*, *cot*, w_0)}
Complejidad: $\Theta(copy(a))$
Descripción: actualiza la cotización del título cuyo nombre es *nt* a la cotización *cotizacion* y ejecuta las promesas de venta y compra que puedan hacerse dada la nueva cotización del título.

AGREGARPROMESA(**in/out** w: `wolfie`, **in** *cliente*: `cliente`, **in** *nomTit*: `string`, **in** *tipo*: `string`, **in** *umbral*: `nat`, **in** *cantidad*: `nat`)
Pre $\equiv$ {$w =_{\text{obs}} w_0 \wedge (\exists$ t:título$)\ (t \in \text{títulos}(w_0) \wedge \text{nombre}(t) = \text{nomTit}) \wedge c \in \text{clientes}(w_0) \wedge_{\text{L}} (\forall p : promesa)(p \in$ promesasDe(c, w_0) $\Rightarrow$ ($nomTit \neq$ título(p) $\vee$ *tipo* $\neq$ tipo(p)) $\wedge$ (tipo = vender $\Rightarrow$ accionesPorCliente(c, título(p)) $\geq$ cantidad(p))) }
Post $\equiv$ {$w =_{\text{obs}}$ agregarPromesa(c, crearPromesa(*nomTit*, *tipo*, *umbral*, *cantidad*), w_0)}
Complejidad: $\Theta(copy(a))$
Descripción: agrega una promesa de tipo *tipo* al cliente c sobre el título cuyo nombre sea *nomTit*.

CLIENTES(**in** w: `wolfie`) $\rightarrow$ *res* : `itDiccClientes(nat, infoCliente)`

Pre $\equiv \{true\}$
Post $\equiv \{\text{haySiguiente}(res) \land \text{esPermutación}(\text{SecuSuby}(res), \text{clientes}(w))\}$
Complejidad: $\Theta(copy(a))$
Descripción: devuelva un iterador a el diccionario de clientes.

TÍTULOS(**in** w : `wolfie`) $\rightarrow res$: `itDiccTítulos(string, infoTitulo)`
Pre $\equiv \{true\}$
Post $\equiv \{\text{haySiguiente}(res) \land \text{esPermutación}(\text{SecuSuby}(res), \text{títulos}(w))\}$
Complejidad: $\Theta(copy(a))$
Descripción: devuelva un iterador a el diccionario de títulos.

PROMESASDE(**in** w : `wolfie`, **in** c : `cliente`) $\rightarrow res$: `itLst(promesasTítulo)`
Pre $\equiv \{c \in \text{clientes}(w)\}$
Post $\equiv \{res =_{\text{obs}} \text{promesasDe}(c, w)\}$
Complejidad: $\Theta(copy(a))$
Descripción: devuelve todas las promesas del cliente c.

ACCIONESPORCLIENTE(**in** w : `wolfie`, **in** $nomTit$: `string`, **in** $cliente$: `c`) $\rightarrow res$: `nat`
Pre $\equiv \{c \in \text{clientes}(w) \land (\exists t : ttulo)(t \in \text{títulos}(w) \land \text{nombre}(t) = nomTit)\}$
Post $\equiv \{res =_{\text{obs}} \text{accionesPorCliente}(c, nomTit, w)\}$
Complejidad: $\Theta(copy(a))$
Descripción: devuelve la cantidad de acciones que tiene el cliente c del título cuyo nombre es $nomTit$.

ENALZA(**in** w : `wolfie`, **in** $nomTit$: `string`) $\rightarrow res$: `bool`
Pre $\equiv \{(\exists t : ttulo)(t \in \text{títulos}(w) \land \text{nombre}(t) = nomTit)\}$
Post $\equiv \{res =_{\text{obs}} \text{enAlza}(nomTit, w)\}$
Complejidad: $\Theta(copy(a))$
Descripción: devuelve *true* si el título acaba de agregarse a Wolfie o si la cotización actual es mayor a la anterior.

Representación

Representación de Wolfie

$wolfie$ **se representa con** `wolfieEstr`

donde `wolfieEstr` es `tupla`(*clientes*: `DiccionarioClientes(cliente, infoCliente)`, *títulos*: `DiccionarioTítulos(nombre, infoTítulo)`, *promesasDe*: `infoPromesas`)

donde `infoPromesas` es `tupla`(*cliente*: `nat`, *actualizado*: `bool`, *promesas*: `lst(promesaTítulo)`)

donde `promesaTítulo` es `tupla`(*nomTit*: `string`, *tipo*: `string`, *umbral*: `nat`, *cantidad*: `nat`)

donde `infoTítulo` es `tupla`(*maxAcciones*: `nat`, *accionesDisponibles*: `nat`, *cotización*: `nat`, *enAlza*: `bool`)

donde `infoCliente` es `tupla`(*títulos*: `DiccionarioTítulos(nombre, infoTítuloCliente)` , *totalAcciones*: `nat`)

donde `infoTítuloCliente` es `tupla`(*cantidadAcciones*: `nat`, *promesas*: `promesas`)

donde `promesas` es `tupla`(*compra*: `promesa` , *venta*: `promesa`)

donde `promesa` es `tupla`(*pendiente*: `bool`, *umbral*: `nat`, *cantidad*: `nat`)

donde cliente es nat

Invariante de representación

Entre wolfieEstr.clientes y wolfieEstr.Títulos:

1. Todos los títulos que están definidos en los infoCliente.títulos también están definidos en el wolfieEstr.títulos.
   $(\forall claveCliente : cliente)$ (def?($claveCliente$, $w.clientes$) $\Rightarrow$ $(\forall nomTit : titulo)$
   (def?($nomTit$, dameTítulos($claveCliente$, $w.clientes$)) $\Rightarrow$ def?($nomTit$, $w.titulos$)))
2. Para cada título definido en wolfieEstr.títulos el infoTítulo.accionesDisponibles es igual a la resta entre: infoTítulo.maxAcciones y la suma de la cantidad de acciones de ese título que tienen todos los clientes, es decir, la suma

de los infoTítuloCliente.cantidadAcciones que se correspondan con el nombre del título que estamos calculando de todos los clientes.
($\forall nomTit : string$) (def?($nomTit$, $w.titulos$) $\Rightarrow$ dameDisponibles($nomTit$, $w.titulos$) = dameMaxAcciones($nomTit$, $w.titulos$) - sumatoriaAccionesTítulo(t, $w.clientes$))

Adentro de wolfieEstr.títulos:

1. accionesDisponibles no puede ser mayor a maxAcciones.
($\forall nomTit : titulo$) (def?($nomTit$, $w.titulos$) $\Rightarrow$ cantidadMáximaAcciones($nomTit$, $w.titulos$) $\geq$ accionesDisponibles($nomTit$, $w.titulos$))

Adentro de wolfieEstr.clientes:

1. En cada infoCliente el totalAcciones tiene que ser igual a la suma de cantidadAcciones de todos los títulos definidos en infoCliente.títulos.
($\forall claveCliente : cliente$) (def?($claveCliente$, $w.clientes$) $\Rightarrow$ totalAcciones($claveCliente$, $w.clientes$) = sumatoriaCantidadAcciones(dameTítulos($claveCliente$, $w.clientes$))
2. En todas las entradas de infoTítuloCliente si promesas.venta.pendiente es verdadero entonces promesas.venta.cantidad tiene que ser menor o igual a el infoTítuloCliente.cantidadAcciones.
($\forall claveCliente : cliente$) (def?($claveCliente$, $w.clientes$) $\Rightarrow$ ($\forall nomTit : titulo$)
(def?($nomTit$, dameTítulos($claveCliente$, $w.clientes$)) $\Rightarrow$
cantidadPrometidasVenta(obtener($nomTit$, dameTítulos($claveCliente$, $w.clientes$)) $\leq$
cantidadAcciones(obtener($nomTit$, dameTítulos($claveCliente$, $w.clientes$))

Adentro de wolfieEstr.promesasDe cuando promesasDe.actualizado sea verdadero:

1. no puede haber más de una promesa de compra sobre cada título
wolfieEstr.promesasDe.actualizado $\Rightarrow$ ($\forall nomTit : string$)
(cantidadDeCompra(wolfieEstr.promesasDe.promesas, nomTit) = 1)
2. no puede haber más de una promesa de venta sobre cada título
wolfieEstr.promesasDe.actualizado $\Rightarrow$ ($\forall nomTit : string$)
(cantidadDeVenta(wolfieEstr.promesasDe.promesas, nomTit) = 1)

Entre wolfieEstr.promesasDe y wolfieEstr.clientes cuando wolfieEstr.promesasDe.actualizado sea verdadero:

1. promesasDe.cliente pertenece a los clientes de wolfie.
wolfieEstr.promesasDe.actualizado $\Rightarrow$ def?(wolfieEstr.clientes, promesasDe.cliente)
2. todas las promesas en promesasDe.promesas están en el correspondiente infoCliente y viceversa.
promesasDe.actualizado $\Rightarrow$
esPermutacion(promesasALista(dameTítulosCliente(wolfieEstr.clientes, promesasDe.cliente)), promesasDe.promesas)

Rep : wolfie $\longrightarrow$ bool

Rep(w) $\equiv$ true $\iff$ ($\forall nomTit : titulo$) (def?($nomTit$, $w.titulos$) $\Rightarrow$ cantidadMáximaAcciones($nomTit$, $w.titulos$) $\geq$ accionesDisponibles($nomTit$, $w.titulos$)) $\wedge$ ($\forall claveCliente : cliente$) (def?($claveCliente$, $w.clientes$) $\Rightarrow$ totalAcciones($claveCliente$, $w.clientes$) = sumatoriaCantidadAcciones(dameTítulos($claveCliente$, $w.clientes$)) $\wedge$ ($\forall nomTit : titulo$) (def?($nomTit$, dameTítulos($claveCliente$, $w.clientes$)) $\Rightarrow$ cantidadPrometidasVenta(obtener($nomTit$, dameTítulos($claveCliente$, $w.clientes$)) $\leq$ cantidadAcciones(obtener($nomTit$, dameTítulos($claveCliente$, $w.clientes$)) $\wedge$ def?($nomTit$, $w.titulos$))) $\wedge$ ($\forall nomTit : string$) (def?($nomTit$, $w.titulos$) $\Rightarrow$ dameDisponibles($nomTit$, $w.titulos$) = dameMaxAcciones($nomTit$, $w.titulos$) - sumatoriaAccionesTítulo(t, $w.clientes$)) $\wedge$ wolfieEstr.promesasDe.actualizado $\Rightarrow$ (($\forall nomTit : string$) (cantidadDeCompra(wolfieEstr.promesasDe.promesas, nomTit) = 1 $\wedge$ cantidadDeVenta(wolfieEstr.promesasDe.promesas, nomTit) = 1)) $\wedge$ def?(wolfieEstr.clientes, promesasDe.cliente) $\wedge$ esPermutacion(promesasALista(dameTítulosCliente(wolfieEstr.clientes, promesasDe.cliente)), promesasDe.promesas)

sumatoriaAccionesTítulo : t $string$ × clientes $dict(cliente, infoCliente)$ ⟶ nat
sumatoriaAccionesTítulo(t,$clientes$) ≡ sumatoriaAccionesTítuloConj(t, claves($clientes$), $clientes$)

sumatoriaAccionesTítuloConj : t $string$ × claves $conj(string)$ × clientes $dict(nat, infoCliente)$ ⟶ nat
sumatoriaAccionesTítuloConj(t, $claves$, $clientes$) ≡ **if** ∅?(claves) **then**
0
else
dameCantAcciones(t, dameTítulos(obtener(dameUno($claves$), $clientes$))) + sumatoriaAccionesTítuloConj(t, sinUno($claves$), $clientes$)
fi

dameCantAcciones : nomTit $string$ × títulos $dict(nat, infoTituloCliente)$ ⟶ nat
dameCantAcciones($nomTit$,$ttulos$) ≡ **if** def?($nomTit$, $titulos$) **then**
obtener($nomTit$, $titulos$).cantidadAcciones
else
0
fi

dameMaxAcciones : nomTit $string$ × títulos $dict(string, infoTitulo)$ ⟶ nat {def?($nomTit$, $titulos$)}
dameMaxAcciones($nomTit$,$ttulos$) ≡ $\prod_1$(obtener($nomTit$, $titulos$))

cantidadMáximaAcciones : nomTit $string$ × títulos $dict(string, infoTitulo)$ ⟶ nat {def?($nomTit$, $titulos$)}
cantidadMáximaAcciones($nomTit$,$ttulos$) ≡ obtener($nomTit$, $titulos$).maxAcciones

accionesDisponibles : nomTit $string$ × títulos $dict(string, infoTitulo)$ ⟶ nat {def?($nomTit$, $titulos$)}
accionesDisponibles($nomTit$,$ttulos$) ≡ obtener($nomTit$, $titulos$).accionesDisponibles

dameTítulos : c nat × clientes $dict(nat, infoCliente)$ ⟶ dict {def?(c, $clientes$)}
dameTítulos(c, $clientes$) ≡ obtener(c, $clientes$).titulos

totalAcciones : c nat × clientes $dict(nat, infoCliente)$ ⟶ dict {def?(c, $clientes$)}
totalAcciones(c, $clientes$) ≡ obtener(c, $clientes$).totalAcciones

sumatoriaCantidadAcciones : títulos $dict(string, infoTtuloCliente)$ ⟶ nat
sumatoriaCantidadAcciones($titulos$) ≡ sumatoriaPrimeraComponenteDiccionario(claves($titulos$), $titulos$)

sumatoriaPrimeraComponenteDiccionario : c $conj(string)$ × d $dict(string, infoTtuloCliente)$ ⟶ nat
sumatoriaPrimeraComponenteDiccionario(c, d) ≡ **if** ∅?(c) **then**
0
else
$\prod_1$(obtener(dameUno(c), d)) + sumatoriaPrimeraComponenteDiccionario(sinUno(c), d)
fi

cantidadPrometidasVentas : t $infoTituloCliente$ ⟶ nat
cantidadPrometidasVentas(t) ≡ t.venta.cantidad

cantidadAcciones : t $infoTituloCliente$ $\longrightarrow$ nat
cantidadAcciones(t) $\equiv$ t.cantidadAcciones

cantidadDeCompra : promesas $secu(promesaTtulo)$ × nomTit $string$ $\longrightarrow$ nat
cantidadDeCompra($promesas, nomTit$) $\equiv$ **if** vacia?(promesas) **then**
0
else
(**if** prim(promesas).nomTit = nomTit ∧ prim(promesas).tipo = compra **then**
1
else
0
fi) + cantidadDeCompra(fin($promesas$), $nomTit$)
fi

cantidadDeVenta : promesas $secu(promesaTtulo)$ × nomTit $string$ $\longrightarrow$ nat
cantidadDeVenta($promesas, nomTit$) $\equiv$ **if** vacia?(promesas) **then**
0
else
(**if** prim(promesas).nomTit = nomTit ∧ prim(promesas).tipo = venta **then**
1
else
0
fi) + cantidadDeVenta(fin($promesas$), $nomTit$)
fi

dameTítulosCliente : clientes $dict(nat$ × infoCliente) × c $cliente$ $\longrightarrow$ dictTítulos
dameTítulosCliente($clientes, c$) $\equiv$ obtener($clientes$, c).títulos

promesasALista : titulos $dict(string$ × infoTítuloCliente) $\longrightarrow$ secu(promesaTítulo)
promesasALista($titulos$) $\equiv$ promesasAListaConClaves(claves($titulos$), $titulos$)

promesasAListaConClaves : claves $conj(string)$ × titulos $dict(string$ × infoTítuloCliente) $\longrightarrow$ secu(promesaTítulo)
promesasAListaConClaves($claves, titulos$) $\equiv$ **if** ∅?(claves) **then**
<>
else
generarPromesasTítulo(dameUno($claves$), obtener($titulos$, dameUno($claves$)) & promesasAListaConClaves(sinUno(claves), $titulos$)
fi

generarPromesasTítulo : nomTit $string$ × info $infoTtuloCliente$ $\longrightarrow$ secu(promesaTítulo)
generarPromesasTítulo($nomTit, info$) $\equiv$ (**if** $info.promesas.compra.pendiente$ **then**
generarPromesaTítulo(nomTit,compra, $info.promesas.compra$)
else
<>
fi) & (**if** $info.promesas.venta.pendiente$ **then**
generarPromesaTítulo(nomTit, venta, $info.promesas.venta$)
else
<>
fi)

generarPromesaTítulo : nomTit *string* × tipo *string* × p *promesa* ⟶ promesaTítulo

generarPromesasTítulo($nomTit$, $tipo$, p) ≡ <$nomTit$, $tipo$, $p.umbral$, $p.cantidad$>

Función de abstracción

Abs : wolfie e ⟶ wolfie {Rep(e)}

Abs(e) ≡ clientes(w) = claves(we.clientes) ∧ títulos(w) = claves($we.titulos$) ∧ ($\forall c : cliente, t : titulo$) ($c \in$ clientes(w) ∧ $t \in$ titulos(w) ⇒ accionesPorCliente(c, nombre(t), w) = dameCantAcciones(nombre(t), dameTítulos(c, $we.clientes$))) ∧ ($\forall c : cliente$) ($c \in$ clientes(w) ⇒ ($\forall p : promesa$) ($p \in$ promesasDe(c, w) ⇔ ($\exists$ $pEstr$: $promesaTtulo$ / $pEstr \in$ promesasAConj(c, $we.clientes$) ∧ $tp.tipo$ = tipo(p) ∧ $tp.umbral$ = limite(p) ∧ $tp.cantidad$ = cantidad(p) ∧ $tp.nomTit$ = titulo(p))))

promesasAConj : c *cliente* × clientes $dict(nat, infoCliente)$ ⟶ conj(tPromesa)

promesasAConj(c, $clientes$) ≡ damePromesas(obtener($clientes$, c).$titulos$)

damePromesas : titulos $dict(string \times$ infoTítuloCliente) ⟶ conj(promesaTítulo)

damePromesas($titulos$) ≡ promesasAConjConClaves(claves($titulos$), $titulos$)

promesasAConjConClaves : claves $conj(string)$ × titulos $dict(string \times$ infoTítuloCliente) ⟶ secu(promesaTítulo)

promesasAConjConClaves($claves$, $titulos$) ≡ **if** ∅?(claves) **then**
∅
else
generarPromesasTítulo(dameUno($claves$), obtener($titulos$, dameUno($claves$)) ∪ promesasAConjConClaves(sinUno(claves), $titulos$)
fi

generarPromesasTítulo : nomTit *string* × info $infoTtuloCliente$ ⟶ secu(promesaTítulo)

generarPromesasTítulo($nomTit$, $info$) ≡ (**if** $info.promesas.compra.pendiente$ **then**
generarPromesaTítulo(nomTit,compra, $info.promesas.compra$)
else
∅
fi) ∪ (**if** $info.promesas.venta.pendiente$ **then**
generarPromesaTítulo(nomTit, venta, $info.promesas.venta$)
else
∅
fi)

generarPromesaTítulo : nomTit *string* × tipo *string* × p *promesa* ⟶ promesaTítulo

generarPromesasTítulo($nomTit$, $tipo$, p) ≡ <$nomTit$, $tipo$, $p.umbral$, $p.cantidad$>

dameCantAcciones : nomTit *string* × títulos $dict(string, infoTituloCliente)$ ⟶ nat

dameCantAcciones($nomTit$,$titulos$) ≡ **if** def?($nomTit$, $titulos$) **then** $\prod_1$(obtener($nomTit$, $titulos$)) **else** 0 **fi**

dameTítulos : c *nat* × clientes $dict(nat, infoCliente)$ ⟶ dict(string, infoTituloCliente) {def?(c, $clientes$)}

dameTítulos(c, $clientes$) ≡ $\prod_1$(obtener(c, $clientes$))

Algoritmos

*i*InaugurarWolfie(**in** *clientes* : `conj(cliente)`)

```
diccTítulos w.títulos ← NuevoDiccionario();                          // O( 1 )
diccClientes w.clientes ← Vacío();                                   // O( 1 )
itConj itClientes = crearIt(clientes);                               // O( 1 )
while haySiguiente(itClientes) do
    ;                                                          // O(#(clientes))
    diccTítulos títulos ← NuevoDiccionario();                        // O( 1 )
    Definir(w.clientes, siguiente(itClientes), <títulos, 0>);        // O( 1 )
    avanzar(itClientes);                                             // O( 1 )
end
```

*i*AgregarTítulo(**in/out** *w* : `wolfie`, **in** *nomTit* : `string`, **in** *maxAcciones* : `nat`, **in** *cot* : `nat`)

```
nat accionesDisponibles ← maxAcciones
bool enAlza ← true
tupla infoTítulo ← <maxAcciones, accionesDisponibles, cot, enAlza>
Definir(nomTit, infoTítulo, w.títulos);                          // O( |nomTit| )
```

```
iActualizarCotización(in/out w: wolfie, in nomTit: string, in cot: nat)
w.promesasDe.actualizado ← false
tupla infoTítulo ← Obtener(w.títulos, nomTit);                                  // O( |nomTit| )
if infoTítulo.cotización >cot then
    infoTítulo.enAlza ← false
end
else
    infoTítulo.enAlza ← true
end
infoTítulo.cotización ← cot
// ejecutamos todas las promesas de venta
itDiccClientes itClientes ← crearIt(w.clientes)
while haySiguiente(itClientes) do
    ;                                                                           // O( #clientes )
    tupla infoCliente ← siguienteSignificado(itClientes)
    if definido(nomTit, infoCliente.títulos) then
        ;                                                                       // O( |nomTit| )
        tupla títutloActual ← obtener(infoCliente.títulos, nomTit);             // O( |nomTit| )
        nat accionesVendidas ← ejecutarVenta(títuloActual.promesas, cot);       // O( 1 )
        if accionesVendidas >0 then
            títuloActual.cantidadAcciones ← títuloActual.cantidadAcciones - accionesVendidas
            infoCliente.totalAcciones ← infoCliente.totalAcciones - accionesVendidas
            infoTítulo.accionesDisponibles ← infoTítulo.accionesDisponibles + accionesVendidas
        end
    end
end
//generamos un arregloOrdenado de tuplas <cliente, totalAcciones>ordenado por la segunda
//componente de las tuplas
itDiccClientes itClientes ← crearIt(w.clientes);                                // O( 1 )
nat cantidadClientes ← #claves(w.clientes)
arr clientesPorAcciones ← crearArreglo(cantidadClientes);                       // O( #clientes )
nat i ← 0
while haySiguiente(itClientes) do
    ;                                                                           // O( #clientes )
    tupla clienteTotalAcciones ← <siguienteClave(itClientes), siguienteSignificado(itClientes).totalAcciones>
end
clientesPorAcciones ← mergeSort(clientesPorAcciones);                           // O( 1 )
// ejecutamos todas las promesas de compra
for nat i ← 0 to (cantidadClientes-1) do
    tupla infoCliente ← obtener(w.clientes, clientesPorAcciones[i])
    if definido(nomTit, infoCliente.títulos) then
        tupla títuloActual ← obtener(infoClientes.títulos, nomTit)
        nat accionesCompradas ← ejecutarCompra(títuloActual.promesas, cot)
        if accionesCompradas >0 then
            títuloActual.cantidadAcciones ← títuloActual.cantidadAcciones + accionesCompradas
            infoCliente.totalAcciones ← infoCliente.totalAcciones + accionesCompradas
            infoTítulo.accionesDisponibles ← infoTítulo.accionesDisponibles - accionesCompradas
        end
    end
end
```

*i*EjecutarVenta(**in/out** *promesas* : `promesas`,**in** *cot* : `nat`) ⟶ res:nat
if *promesas.venta.pendiente* $\wedge_{\text{L}}$ *promesas.venta.umbral* <*cot* **then**
    promesas.venta.pendiente ← false
    res ← promesas.venta.cantidad
    else
        res ← 0
    end
end

*i*EjecutarCompra(**in/out** *promesas* : `promesas`,**in** *cot* : `nat`) ⟶ res:nat
if *promesas.compra.pendiente* $\wedge_{\text{L}}$ *promesas.compra.umbral* <*cot* **then**
    promesas.compra.pendiente ← false
    res ← promesas.compra.cantidad
    else
        res ← 0
    end
end

*i*AgregarPromesa(**in/out** *w* : `wolfie`,**in** *c* : `cliente`, **in** *nomTit* : `string`, **in** *tipo* : `string`, **in** *umbral* : `nat`, **in** *cantidad* : `nat`)

if *c* = *w.promesasDe.cliente* **then**
    w.promesasDe.actualizado ← false
end

tupla infoCliente ← obtener(c, w.clientes)
if *definido(infoCliente.títulos, nomTit)* **then**
    tupla infoTítuloCliente ← obtener(infoCliente.títulos, nomTit)
    if *tipo* = *venta* **then**
        infoTítuloCliente.promesas.venta ← <true, umbral, cantidad>
    end
    if *tipo* = *compra* **then**
        infoTítuloCliente.promesas.compra ← <true, umbral, cantidad>
    end
end
else
    tupla infoTítuloCliente
    infoTítuloCliente.cantidadAcciones ← 0
    if *tipo* = *venta* **then**
        infoTítuloCliente.promesas.compra ← <false, 0, 0>
        infoTítuloCliente.promesas.venta ← <true, umbral, cantidad>
    end
    if *tipo* = *compra* **then**
        infoTítuloCliente.promesas.venta ← <false, 0, 0>
        infoTítuloCliente.promesas.compra ← <true, umbral, cantidad>
    end
    definir(infoCliente.títulos, nomTit, infoTítuloCliente)
end

*i*clientes() ⟶ res: itDiccClientes(nat)

res ← crearItDiccOrd(w.clientes)

*i*títulos() ⟶ res: itDiccTítulos(nat)

res ← crearItDiccTítulos(w.títulos)

*i*PromesasDe(**in/out** w : `wolfie`, **in** c : `cliente`) $\longrightarrow$ res: itLst(promesasTítulo)

if c = *w.promesasDe.cliente* $\wedge$ *w.promesasDe.actualizado* **then**
  res $\leftarrow$ crearIt(w.promesasDe.promesas)
end

tupla infoCliente $\leftarrow$ obtener(w.clientes, c)
itDiccTítulos itTítulos $\leftarrow$ crearItDiccTítulos(infoCliente.títulos)
lista promesas $\leftarrow$ vacia()
while *haySiguiente(itTítulos)* **do**
  tupla infoTítulo $\leftarrow$ siguienteSignificado(itTítulos)
  if *infoTítulo.promesas.venta.pendiente* **then**
    agregarAdelante(promesas, <siguienteClave(itTítulos), venta, infoTítulo.promesas.venta.umbral, infoTítulo.promesas.venta.cantidad>
  end
  if *infoTítulo.promesas.compra.pendiente* **then**
    agregarAdelante(promesas, <siguienteClave(itTítulos), compra, infoTítulo.promesas.compra.umbral, infoTítulo.promesas.compra.cantidad>
  end
end

w.promesasDe.actualizado $\leftarrow$ true
w.promesasDe.cliente $\leftarrow$ c
w.promesasDe.promesas $\leftarrow$ promesas

res $\leftarrow$ crearIt(promesas)

*i*AccionesPorCliente(**in** w : `wolfie`, **in** *nomTit* : `string`, **in** c : `cliente`) $\longrightarrow$ res: nat

tupla infoCliente $\leftarrow$ obtener(w.clientes, c)
tupla infoTítulo $\leftarrow$ obtener(infoCliente.títulos, nomTit)
res $\leftarrow$ infoTítulo.cantidadAcciones

*i*EnAlza(**in** w : `wolfie`, **in** *nomTit* : `string`) $\longrightarrow$ res: bool

tupla infoTítulo $\leftarrow$ obtener(w.títulos, nomTit)
res $\leftarrow$ infoTítulo.enAlza)

3. Módulo Diccionario Títulos

Interfaz

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

se explica con: DICCIONARIO(STRING, α).

géneros: `diccTítulo(`*String*`,`α`)`, `itDicc(`α`)`.

Operaciones básicas de diccionario títulos

NUEVODICCIONARIO() $\rightarrow$ *res* : `diccTítulo(`*String*`,`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacio()}
Complejidad: $\Theta(1)$

Descripción: Crea un nuevo diccionario vacío.

DEFINIR(**in** c: `string`, **in** s: α, **in/out** d: `diccTítulo(`$String$`,`α`)`)
Pre $\equiv$ $\{d =_{\text{obs}} d_0\}$
Post $\equiv$ $\{d =_{\text{obs}} \text{definir}(d_0, c, s)\}$
Complejidad: $\Theta(|c| + copy(s))$
Descripción: Define la clave c con el significado s en el diccionario d.
Aliasing: Se agrega por copia el significado s

OBTENER(**in** c: `string`, **in** d: `diccTítulo(`$String$`,`α`)`) $\rightarrow$ res : α
Pre $\equiv$ $\{\text{def?}(c, d)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{obtener}(c, d)\}$
Complejidad: $\Theta(|c|)$
Descripción: Devuelve el significado de la clave c contenida en el diccionario d.

DEFINIDO?(**in** c: `string`, **in** d: `diccTítulo(`$String$`,`α`)`) $\rightarrow$ res : $bool$
Pre $\equiv$ $\{\text{true}\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{def?}(c, d)\}$
Complejidad: $\Theta(|c|)$
Descripción: Chequea si está definida la clave c en el diccionario d.

#CLAVES(**in** d: `diccTítulo(`$String$`,`α`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{\text{true}\}$
Post $\equiv$ $\{res =_{\text{obs}} \#(\text{claves}(d))\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve la cantidad de claves del diccionario.

Operaciones del iterador

CREARIT(**in** d: `diccTítulo(`$String$`,`α`)`) $\rightarrow$ res : `itDicc(`$String$`,`α`)`
Pre $\equiv$ $\{\text{true}\}$
Post $\equiv$ $\{res =_{\text{obs}} <\text{crearItUni}(<>,d.\text{claves}), \text{crearItUni}(<>,d.\text{alfas})>\}$
Complejidad: $\Theta(1)$
Descripción: Crea un iterador unidireccional del diccionario. Se pueden recorrer los elementos aplicando iterativamente Siguiente

HAYSIGUIENTE?(**in** it: `itDicc(`$String$`,`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{\text{true}\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{haySiguiente?}(it)\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar

SIGUIENTE(**in** it: `itDicc(`$String$`,`α`)`) $\rightarrow$ res : `tupla(`$String$`, `α`)`
Pre $\equiv$ $\{\text{HaySiguiente?}(it)\}$
Post $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{Siguiente}(it))\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve el elemento siguiente del iterador
Aliasing: res.significado es un puntero al objeto α y es modificable si y sólo si it es modificable. En cambio, res.clave no es modificable

SIGUIENTECLAVE(**in** it: `itDicc(`$String$`,`α`)`) $\rightarrow$ res : `String`
Pre $\equiv$ $\{\text{HaySiguiente?}(it)\}$
Post $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{Siguiente}(it).\text{clave})\}$
Complejidad: $\Theta(1)$
Descripción: Devuelve la clave del elemento siguiente del iterador
Aliasing: res no es modificable

SIGUIENTESIGNIFICADO(**in** it: `itDicc(`$String$`,`α`)`) $\rightarrow$ res : α
Pre $\equiv$ $\{\text{HaySiguiente?}(it)\}$
Post $\equiv$ $\{\text{alias}(res =_{\text{obs}} \text{Siguiente}(it))\}$
Complejidad: $\Theta(1)$

Descripción: Devuelve el significado del elemento siguiente del iterador
Aliasing: *res* es modificable si y sólo si *it* es modificable

AVANZAR(**in/out** *it* : `itDicc(`*String*`,`α`)`)
Pre $\equiv \{it = it_0 \wedge \text{HaySiguiente?}(it)\}$
Post $\equiv \{it =_{\text{obs}} \text{Avanzar}(it_0)\}$
Complejidad: $\Theta(1)$
Descripción: Avanza a la posición siguiente del iterador

Representación

Representación del diccionario:

`diccTítulo(`*String*`,` α`)` **se representa con** `estr_diccTítulo`

donde `estr_diccTítulo` es `tupla(`*raíz*: `estr_nodo` , *Claves*: `lista(`*String*`)` , *alfas*: `lista(punteros(`α`))`)

donde `estr_nodo` es `tupla(`*significado*: `puntero` (α) , *hijos*: `array[256] de puntero(estr_nodo)`)

Invariante de representación:

1. Sin ciclos en el arbol
2. Las Hojas no pueden tener significado nulo
3. La cantidad de claves ingresadas en e.claves debe ser igual a la cantidad de significados válidos (distintos de NULL) del árbol Trie y a la cantidad de punteros ingresados en e.alfas
4. Las claves contenidas en *e*.claves deben estar definidas en el arbol Trie
5. Todos los punteros de significados válidos del arbol Trie deben estar definidos en *e*.alfas

Rep : estr_diccTítulo $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$
SinCiclos(e.raíz, $\emptyset$) $\wedge_{\text{L}}$ 1.
SignificadosHojasNotNull(e.raíz) $\wedge$ 2.
`Longitud(`e`.Claves)` $=_{\text{obs}}$ `Longitud(`e`.alfas)` $\wedge$
`Longitud(`e`.alfas)` $=_{\text{obs}}$ CantidadSignificados(e.raíz) $\wedge$ 3.
$(\forall clave : String)$ ((está?($clave$, e.claves)) $\Leftrightarrow$ definido?($clave$,e.raíz)) $\wedge$ 4.
$(\forall p : puntero(\alpha)$ (está?(p, e.alfas)) $\Rightarrow$ ($\exists$ $clave : String$) está?($clave$,e.claves) $\wedge$
p = ObtDeEstruc($clave$, e.raíz)) 5.

SignificadoHojaNotNull : estr_nodo e $\longrightarrow$ bool

SignificadoHojaNotNull $\equiv$ **if** Hoja?(e, 0) **then** $\neg$(significado(e)$=_{\text{obs}}$ NULL) **else** RecorrerHijos(e,0) **fi**

Hoja? : estr_nodo e $\times$ nat $\longrightarrow$ bool

Hoja?(e,n) $\equiv$ (e.hijos[n] $=_{\text{obs}}$ NULL) $\wedge$ **if** (n <256) **then** Hoja?(e,n+1) **else** true **fi**

RecorrerHijos : estr_nodo e $\times$ nat $\longrightarrow$ bool {SinCiclos(e,0)}

RecorrerHijos(e,n) $\equiv$ (**if** ($\neg$(e.hijos[n] $=_{\text{obs}}$ NULL)) **then** SignificadoHojaNotNull(*(e.hijos[n])) **else** true **fi**) $\wedge$
(**if** (n <256) **then** RecorrerHijos(e,n+1) **else** true **fi**) FI

CantSignificados : estr_nodo e $\longrightarrow$ nat

CantSignificados(e) $\equiv$ (**if** (e.significado $=_{\text{obs}}$ NULL) **then** 0 **else** 1 **fi**) + SigHijos(e,0)

SigHijos : estr_nodo e $\times$ nat $\longrightarrow$ nat

SigHijos(e,n) $\equiv$ (**if** (e.hijos[n] $=_{\text{obs}}$ NULL) **then** 0 **else** CantSignificados(*(e.hijos[n])) **fi**) + (**if** (n <256) **then** SigHijos(e,n+1) **else** 0 **fi**)

Función de abstracción:

Abs : estr_diccTítulo e $\longrightarrow$ dicc(string, α) {Rep(e)}

Abs(e) $=_{obs}$ d: dicc(string, α) | #(claves(d)) $=_{obs}$ `Longitud(e.clave)` $\wedge_L$ 1.
(∀c: string)(def?(c, d) $\Rightarrow_L$ 2.
(definido?(c, e.raíz) $\wedge_L$
obtener(c, d) $=_{obs}$ *(ObtDeEstruc(c, e.raíz))))

definido? : string e × estr_nodo e $\longrightarrow$ *bool*

definido?(c,n) $\equiv$ **if** vacía?(c) **then**
¬(n.significado = NULL)
else
if n.hijos[ORD(prim(c)] = NULL **then** false **else** definido?(fin(c),n.hijos[ORD(prim(c))] **fi**
fi

ObtDeEstruc : string e × estr_nodo e $\longrightarrow$ α

ObtDeEstruc(c,n) $\equiv$ **if** vacía?(c) **then**
n.significado
else
if n.hijos[ORD(prim(c)] = NULL **then**
NULL
else
ObtDeEstruc(fin(c),n.hijos[ORD(prim(c))]
fi
fi

Representación del iterador:

El iterador del diccionario es simplemente un par de iteradores a las listas correspondientes.
Lo único que hay que pedir es que satisfaga el Rep de este par de listas.
Por implementación, alcanza con que sea unidireccional.

`itDiccTítulos(`*String*`,`α`)` **se representa con** `itDic`

donde `itDic` es `tupla(`*claves*: `itLista(`*String*`)` , *significados*: `itLista(`α`)`)

Rep : itDic $\longrightarrow$ bool

Rep(it) $\equiv$ true $\Longleftrightarrow$
Rep(it.claves) $\wedge$ Rep(it.significados)

Abs : itDic it $\longrightarrow$ itMod(tupla(*String*,α)) {Rep(it)}
Abs(it) $\equiv$ CrearItMod(Join(Siguientes(it.claves), Siguientes(it.significados)))

Join : secu(*String*)a × secu(α)b $\longrightarrow$ secu(tupla(*String*,α)) {long(a) = long(b)}

Join(a,b) $\equiv$ **if** vacía?(a) **then** <> **else** <prim(a), prim(b)>• Join(Fin(a), Fin(b)) **fi**

Algoritmos

iNuevoDiccionario() $\longrightarrow$ res: bool
Orden Complejidad: *O(1)*
res $\leftarrow$ <raíz: iNodoNuevo(), Claves : Vacía(), Alfas : Vacía() >; `// O(1)`

iNodoNuevo() $\longrightarrow$ res: estr_nodo
Orden Complejidad: *O(1)*
res $\leftarrow$ <significado: NULL, hijos:CrearArreglo() >; `// O(1)`
for *var* i : *nat* $\leftarrow$ 0 **to** 255*;* `// O(256)`
do
res.hijos[i] $\leftarrow$ NULL
end

iDefinir(**in** c: `string`, **in** s: α, **in/out** d: `estr_diccTítulo`) $\longrightarrow$ res: estr_dicTrie

Orden Complejidad: *O(|c|)*

AgregarAtrás(d.*Claves*, c) ; // O(|c|)
var *actual* : puntero(*estr_nodo*) ← &(*d*.raíz) ; // O(1)
for *var i* : *nat* ← 0 **to** *(Longitud(c) - 1);* // O(|c|)
do
  if *actual* → *hijos[ORD(c[i])] = NULL* **then**
    actual → hijos[ORD(c[i])] ← &(iNodoNuevo())
  end
  actual ← (actual → hijos[ORD(c[i])])
end
actual → significado ← &(Copiar(*s*)); // O(1)
AgregarAtrás(d.*alfas*, actual → significado) ; // O(1)

*i*Obtener(**in** *c*: string, **in** *d*: estr_diccTítulo) ⟶ res:*α*
Orden Complejidad: *O(|c|)*

var *actual* : puntero(*estr_nodo*) ← &(*d*.raíz); // O(1)
for *var i* : *nat* ← 0 **to** *(Longitud(c) - 1);* // O(|c|)
do
  actual ← (actual → hijos[ORD(c[i])])
end
res ← * (actual → significado); // O(1)

*i*Definido?(**in** *c*: string, **in** *d*: estr_diccTítulo)⟶ res: bool
Orden Complejidad: *O(|c|)*

var *actual* : puntero(*estr_nodo*) ← &(*d*.raíz) ; // O(1)
var i : Nat ← 0 ; // O(1)
while *actual != NULL && (i <Longitud(c));* // O(|c|)
do
  i ← *i* + 1
  actual ← (actual → hijos[ORD(c[i])])
end
if *(actual != NULL)* **then**
  if *(actual* → *significado) != NULL* **then**
    res ← *true*; // O(1)
  else
    res ← *false*; // O(1)
  end
else
  res ← *false*; // O(1)
end

i#Claves(**in** *d*: estr_diccTítulo)⟶ res: nat
Orden Complejidad: *O(1)*

res ← Longitud(*d*.Claves); // O(1)

Algoritmos del iterador

*i*CrearIT(in d: diccTítulo(*String*,*α*)) ⟶ res: ItDicc(*String*,*α*)
Orden Complejidad: *O(1)*

res ← <claves:CrearIt(*d*.claves), significados: CrearIt(*d*.alfas)>

*i*HaySiguiente?(in it: ItDicc(*String*,*α*)) ⟶ res: bool
Orden Complejidad: *O(1)*

res ← HaySiguiente(it.claves)

*i*Siguiente(in it: ItDicc(*String*,*α*)) ⟶ res: tupla(*String*, *α*)

Orden Complejidad: *O(1)*

res ← <Siguiente(it.claves), Siguiente(it.significados) >

*i*SiguienteClave(in it: ItDicc(*String*,α)) ⟶ res: *String*
Orden Complejidad: *O(1)*

res ← Siguiente(it.claves)

*i*SiguienteSignificado(in it: ItDicc(*String*,α)) ⟶ res: puntero(α)
Orden Complejidad: *O(1)*

res ← Siguiente(it.significados)

*i*Avanzar(in it: ItDicc(*String*,α)) ⟶ res: ItDicc(*String*,α)
Orden Complejidad: *O(1)*

res ← <Avanzar(it.claves), Avanzar(it.significados) >

4. Módulo Diccionario Clientes

Interfaz

parámetros formales
géneros α
función COPIAR(**in** a: α) → res : α
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's

se explica con: DICCIONARIO(α,σ), ITERADOR UNIDIRECCIONAL(TUPLA(α,σ)).

géneros: `dictClientes(nat, infoCliente)`, `itDictClientes(Tupla(`α`,`σ`))`.

Operaciones básicas de diccionario ordenado

VACÍO(**in** n: `nat`) → res : `dictClientes(nat, infoCliente)`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} vacio$}
Complejidad: $O(n)$
Descripción: genera un diccionario vacío.

DEFINIR(**in/out** d: `dict(nat, infoCliente)`, **in** c: `nat`, **in** s: `infoCliente`)
Pre ≡ {$d =_{\text{obs}} d_0$}
Post ≡ {$d =_{\text{obs}} definir(d_0, c, s)$}
Complejidad: $O(\#claves(d) + copy(c) + copy(s))$
Descripción: define la clave c con el significado s en el diccionario.
Aliasing: los elementos c y s se definen por copia.

OBTENER(**in** d: `dictClientes(nat, infoCliente)`, **in** c: `nat`) → res : `infoCliente`
Pre ≡ {def?(c,d)}
Post ≡ {esAlias(res, obtener(c, d))}
Complejidad: $O(log(n))$
Descripción: obtiene el significado σ que corresponde a la clave c.
Aliasing: se genera aliasing entre res y el significado σ

Operaciones del iterador

CrearIt(**in** d: `dicc(nat, infoCliente)`) $\rightarrow$ res : `itDicc(`α`,`σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion(SecuSuby(res), d)) $\wedge$ vacia?(Anteriores(res))}
Complejidad: $\Theta(1)$
Descripción: crea un iterador unidireccional del diccionario, de forma tal que HayAnterior evalúe a false (i.e., que se pueda recorrer los elementos aplicando iterativamente Siguiente)

HaySiguiente(**in** it: `itDicc(nat, infoCliente)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ haySiguiente?(it)}
Complejidad: $\Theta(1)$
Descripción: devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar.

Siguiente(**in** it: `itDicc(nat,infoCliente)`) $\rightarrow$ res : `tupla(`α`,`σ`)`
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ Siguiente(it))}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento siguiente del iterador.
Aliasing: res.significado es modificable si y sólo si it es modificable. En cambio, res.clave no es modificable.

SiguienteClave(**in** it: `itDicc(nat, infoCliente)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ SiguienteClave(it))}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento siguiente del iterador.
Aliasing: res.significado es modificable si y sólo si it es modificable. En cambio, res.clave no es modificable.

SiguienteSignificado(**in** it: `itDicc(nat, infoCliente)`) $\rightarrow$ res : `infoCliente`
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ SiguienteSignificado(it))}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento siguiente del iterador.
Aliasing: res.significado es modificable si y sólo si it es modificable. En cambio, res.clave no es modificable.

Avanzar(**in** it: `itDicc(nat, infoCliente)`)
Pre $\equiv$ {HaySiguiente?(it)}
Post $\equiv$ {alias(res $=_{\text{obs}}$ Avanzar(it))}
Complejidad: $\Theta(1)$
Descripción: devuelve el elemento siguiente del iterador.
Aliasing: res.significado es modificable si y sólo si it es modificable. En cambio, res.clave no es modificable.

Representación

Representación de Diccionario Clientes

`dictClientes(nat, infoCliente)` **se representa con** `dc`

donde `dc` es `tupla`(*claves*: `arreglo(nat)`, *significados*: `arreglo(infoCliente)`, *tamanio*: `nat`)

Invariante de representación

- La capacidad de los contenedores de claves y significados debe ser la misma.
- dc.tamanio debe indicar la cantidad de entradas en el diccionario y éstas deben estar en las primeras (dc.tamanio-1) primeras posiciones de los respectivos arreglos.
- El arreglo de claves debe estar ordenado.

Rep : dc $\longrightarrow$ bool

Rep(dc) $\equiv$ true $\iff$ (tam($dc.claves$)) = tam($d.significados$)) $\wedge$
($\forall$ p, q: nat) $p <_{dc.tamanio} \Rightarrow_{\text{L}}$ (definido?($dc.claves$, p) $\wedge$ definido?($dc.significados$, p)) $\wedge$ ($p < q <$ $dc.tamanio \Rightarrow dc.claves[p] <_{dc.claves[p]}$)

Abs : dictClientes(nat, infoCliente) dc $\longrightarrow$ dict(nat, infoCliente) {Rep(dc)}

Abs(dc) $=_{\text{obs}}$ d: dict(nat, infoCliente) | ($\forall c : \alpha$)(def?(c, d) $\Leftrightarrow$ $c \in$ arregloAConjunto($dc.claves$, tam($dc.claves$)) $\wedge$ (def?(c, d) $\Rightarrow$ obtener(d, c) = $dc.significados$[posición($dc.claves$, 0, c)]))

arregloAConjunto : arr *arreglo*(*nat*) $\times$ tamanio *nat* $\longrightarrow$ conj(nat)

arregloAConjunto(arr, tamanio) $\equiv$
if 0?(*tamanio*) **then**
  $\emptyset$
else
  if definido?(arr, tamanio-1) **then**
    Ag(arr[tamanio-1], arregloAConjunto(arr, tamanio-1)
  else
    arregloAConjunto(arr, tamanio-1)
  fi
fi

posicion : arr *arreglo*(*nat*) $\times$ pos *nat* $\times$ buscado *nat* $\longrightarrow$ nat {buscado $\in$ arregloAConjunto(arr, tam(arr))}

posicion(arr, pos, buscado) $\equiv$
if arr[pos] = buscado **then** pos **else** posicion(arr, pos+1, buscado) **fi**

Representación del iterador

`itDictClientes(nat, infoCliente)` **se representa con** `iter`

donde `iter` es `tupla`(*posición*: `nat`, *límite*: `nat`, *claves*: `puntero(arrOrd(nat))`, *significados*: `puntero(arr(infoCliente))`)

Rep : iter $\longrightarrow$ bool

Rep(it) $\equiv$ true $\iff$ iter.posición <iter.límite

Abs : iter it $\longrightarrow$ itUni(α) {Rep(it)}

Abs(it) $=_{\text{obs}}$ b: itUni(α) | Siguientes(b) = arreglosASecuDesde(it.posición, it.límite, it.claves, it.significados)

arreglosASecuDesde : posición *nat* $\times$ límite *nat* $\times$ claves *puntero*(*arrOrd*(*nat*)) $\times$ significados *puntero*(*arrOrd*(*infoClier*

arreglosASecuDesde(posición, límite, claves, significados) $\equiv$
if posición = límite **then**
  <>
else
  <claves[posición], significados[posición]>• arreglosASecuDesde(posición+1, límite, claves, significados)
fi

Algoritmos

*i*vacío(**in** *n*: `nat`) ⟶ res: dictClientes(nat, infoCliente)

dc.claves ← crearArreglo(n)
dc.significados ← crearArreglo(n)
dc.tamanio ← n
res ← dc

*i*definir(**in/out** *dc*: `dictClientes(nat, infoCliente)`, `in` *c*: `nat`, `in` *s*: `infoCliente`)

posActual ← dc.tamanio
while *(dc.claves[posActual-1] >c)* **do**
    dc.claves[posActual] ← dc.claves[posActual-1]
    dc.significados[posActual ← dc.significados[posActual-1]
    posActual–;
end
dc.claves[posActual] ← c
dc.significados[posActual] ← s
dc.tamanio += 1

*i*obtener(**in/out** *dc*: `dictClientes(nat, infoCliente)`, `in` *c*: `nat`) ⟶ res: infoCliente

der ← dc.tamanio-1
izq ← 0
medio ← dc.tamanio/2
while *dc.clave[medio] != c* **do**
    if *dc.clave[medio] >c* **then**
        der ← medio
    end
    holas **if** *dc.clave[medio] <c* **then**
        izq ← medio
    end
end
res ← dc.significados[medio]

*i*crearIt(**in/out** *dc*: `dictClientes(nat, infoCliente)`) ⟶ res: itDictClientes(nat, infoCliente)

it.posicion ← 0
it.limite ← tamanio(dc.significados)
it.claves ← &(dc.claves)
it.significados ← &(dc.significados)
res ← it

*i*haySiguiente(**in** *it*: `itDictClientes(nat, infoCliente)`) ⟶ res: bool

res ← it.posición <it.límite

*i*siguiente(**in/out** *dc*: `itDictClientes(nat, infoCliente)`) ⟶ res: tupla(nat, infoCliente)

res ← <*(it.claves[it.posición]), *(it.significados[it.posición])>

*i*siguienteClave(**in/out** *dc*: `itDictClientes(nat, infoCliente)`) ⟶ res: nat

res ← *(it.claves[it.posición])

*i*siguienteSignificado(**in/out** *dc*: `itDictClientes(nat, infoCliente)`) ⟶ res: infoCliente

res ← *(it.significados[it.posición])

*i*avanzar(**in/out** *dc*: `itDictClientes(nat, infoCliente)`)

it.posición ← it.posición + 1