

# 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

<b>1. Observaciones generales</b>	<b>3</b>
<b>2. Módulo Wolfe</b>	<b>3</b>
<b>3. Módulo Diccionario Títulos</b>	<b>13</b>
<b>4. Módulo Diccionario Clientes</b>	<b>18</b>

## 1. Observaciones generales

Convenciones que adoptamos en todos los módulos:

- nos referimos a los campos de las tuplas por el nombre de los mismos, no por  $\prod_1, \prod_2, \dots, \prod_n$ .
- 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(campoEspecial1: tipo1, campoEspecial2: tipo2, campoEspecial3: tipo3)`  
después en los algoritmos cada vez que usemos una tupla con esos tipos usamos el alias `tuplaEspecial` y nos referimos a sus campos por `campoEspecial1`, `campoEspecial2` y `campoEspecial3`.

### Correcciones realizadas:

#### Diccionario Clientes

- (estructura) `d.claves` dejó de ser un `arregloOrdenado` para ser un `arregloDimensionable`.
- (estructura) agregamos un `dc.tamano` que nos dice cuántas claves hay definidas en el diccionario. Esto es necesario porque para asegurar que obtener toma  $O(\log n)$  necesitamos saber exactamente dónde están las claves en el arreglo. En este caso estarán en las primeras  $[0, dc.tamano)$  posiciones.
- (invariante) agregamos la condición de que `d.claves` debe estar ordenado.
- (invariante) agregamos el invariante escrito en lenguaje coloquial.
- (algoritmos) la búsqueda binaria se realiza en el mismo obtener del diccionario.

#### Diccionario Títulos

- (estructura) sacamos la lista de significados del diccionario.
- (estructura) cambiamos la estructura del iterador porque ahora sólo guardamos una lista de claves y la de significados no.
- (invariante) agregamos la restricción de que dos padres no pueden compartir un hijo.
- (invariante) hicimos las modificaciones necesarias para que la nueva estructura de representación fuera consistente.
- (algoritmos) corregimos el error de agregar claves repetidas al listado de claves lo cual hacía que el iterador recorra repetidos y las complejidades no se satisficieran.
- (algoritmos) el `agregarAtrás(dc.claves, c)` solo lo hacemos en los casos en los que la clave no estaba previamente definida.
- (algoritmos) modificamos todos los algoritmos del iterador para que operen sobre la nueva estructura.
- corregimos un error en la axiomatización de `ObtDeEstruct`, función auxiliar de la función de abstracción, que consideraba el caso en el que no estaba definida la clave que recibía y nunca era llamada de esa forma.

## 2. Módulo Wolfie

### Interfaz

#### parámetros formales

**géneros**      $\alpha$   
**función**    `COPIAR(in a:  $\alpha$ )  $\rightarrow$  res :  $\alpha$`   
**Pre**  $\equiv \{\text{true}\}$   
**Post**  $\equiv \{\text{res} =_{\text{obs}} a\}$   
**Complejidad:**  $\Theta(\text{copy}(a))$   
**Descripción:** función de copia de  $\alpha$ '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**  $\equiv \{res =_{obs} inaugurarWolfie(clientes)\}$

**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**  $\equiv \{w =_{obs} w_0 \wedge (\forall t : titulo) (t \in \text{títulos}(w_0) \Rightarrow \text{nombre}(t) \neq nomTit) \}$

**Post**  $\equiv \{w =_{obs} agregarTitulo(crearTitulo(nomTit, cot, maxAcciones), w_0)\}$

**Complejidad**:  $O(|nomTit|)$

**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**  $\equiv \{w =_{obs} w_0 \wedge (\exists t : titulo) (t \in \text{títulos}(w_0) \wedge \text{nombre}(t) = nomTit)\}$

**Post**  $\equiv \{w =_{obs} actualizarCotizacion(nomTit, cot, w_0)\}$

**Complejidad**:  $O(\#w.clientes * (|nomTit| + \log \#w.clientes))$

**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**  $\equiv \{w =_{obs} w_0 \wedge (\exists t : titulo) (t \in \text{títulos}(w_0) \wedge \text{nombre}(t) = nomTit) \wedge c \in \text{clientes}(w_0) \wedge_L (\forall p : promesa) (p \in \text{promesasDe}(c, w_0) \Rightarrow (nomTit \neq \text{título}(p) \vee tipo \neq \text{tipo}(p)) \wedge (tipo = \text{vender} \Rightarrow \text{accionesPorCliente}(c, \text{título}(p)) \geq \text{cantidad}(p))) \}$

**Post**  $\equiv \{w =_{obs} agregarPromesa(c, crearPromesa(nomTit, tipo, umbral, cantidad), w_0)\}$

**Complejidad**:  $O(|nomTit| + \log \#w.clientes)$

**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**  $\equiv \{\text{haySiguiente}(res) \wedge \text{esPermutación}(\text{SecuSuby}(res), \text{clientes}(w))\}$

**Complejidad**:  $O(1)$

**Descripción**: devuelve un iterador a el diccionario de clientes.

TÍTULOS(**in** *w*: wolfie)  $\rightarrow$  *res* : itDiccTítulos(string, infoTitulo)

**Pre**  $\equiv \{true\}$

**Post**  $\equiv \{\text{haySiguiente}(res) \wedge \text{esPermutación}(\text{SecuSuby}(res), \text{títulos}(w))\}$

**Complejidad**:  $O(1)$

**Descripción**: devuelve un iterador a el diccionario de títulos.

PROMESASDE(**in** *w*: wolfie, **in** *c*: cliente)  $\rightarrow$  *res* : itLst(promesasTitulo)

**Pre**  $\equiv \{c \in \text{clientes}(w)\}$

**Post**  $\equiv \{res =_{obs} \text{promesasDe}(c, w)\}$

**Complejidad**:  $O(\#w.títulos * \max(|nomTit|))$

**Descripción**: devuelve todas las promesas del cliente *c*.

ACCIONESPORCLIENTE(**in** *w*: wolfie, **in** *nomTit*: string, **in** *cliente*: c)  $\rightarrow$  *res* : nat

**Pre**  $\equiv \{c \in \text{clientes}(w) \wedge (\exists t : titulo) (t \in \text{títulos}(w) \wedge \text{nombre}(t) = nomTit)\}$

**Post**  $\equiv \{res =_{obs} \text{accionesPorCliente}(c, nomTit, w)\}$

**Complejidad**:  $O(\log(\#w.clientes) * |nomTit|)$

**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**  $\equiv \{(\exists t : titulo) (t \in \text{títulos}(w) \wedge \text{nombre}(t) = nomTit)\}$

**Post**  $\equiv \{res =_{obs} \text{enAlza}(nomTit, w)\}$

**Complejidad**:  $O(\max(|nomTit|))$

**Descripción**: devuelve *true* si el título acaba de agregarse a Wolfie o si la cotización actual es mayor a la anterior.

MAXIMARACHA(**in**  $w$ : wolfie, **in**  $nomTit$ : string)  $\rightarrow res$ : bool

**Pre**  $\equiv \{(\exists t : ttulo)(t \in \text{títulos}(w) \wedge \text{nombre}(t) = nomTit)\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{maximaRacha}(\text{dameTítulo}(nomTit, w))\}$

**Complejidad:**  $O(\max(|nomTit|))$

**Descripción:** devuelve *true* si el título acaba de agregarse a Wolfie o si la cotización actual es mayor a la anterior.

TITULOMASVOLATIL(**in**  $w$ : wolfie)  $\rightarrow res$ : string

**Pre**  $\equiv \{\neg \emptyset?(w.\text{títulos})\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{títuloMásVolátil}(w)\}$

**Complejidad:**  $O(\#w.\text{títulos} * \max(|nomTit|))$

**Descripción:** devuelve uno de los títulos con más fluctuaciones de wolfie.

## 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,  
*rachaMaxima*: nat, *rachaActual*: nat, *fluctuaciones*: nat)

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*.  
( $\forall \text{claveCliente} : \text{cliente}$ ) ( $\text{def?}(\text{claveCliente}, w.\text{clientes}) \Rightarrow (\forall \text{nomTit} : \text{titulo})$   
( $\text{def?}(\text{nomTit}, \text{dameTítulos}(\text{claveCliente}, w.\text{clientes})) \Rightarrow \text{def?}(\text{nomTit}, w.\text{títulos}))$ )
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.  
( $\forall \text{nomTit} : \text{string}$ ) ( $\text{def?}(\text{nomTit}, w.\text{títulos}) \Rightarrow \text{dameDisponibles}(\text{nomTit}, w.\text{títulos}) =$   
 $\text{dameMaxAcciones}(\text{nomTit}, w.\text{títulos}) - \text{sumatoriaAccionesTítulo}(t, w.\text{clientes})$ )

Adentro de *wolfieEstr.títulos*:

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

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*.

$(\forall \text{claveCliente} : \text{cliente}) (\text{def?}(\text{claveCliente}, w.\text{clientes}) \Rightarrow \text{totalAcciones}(\text{claveCliente}, w.\text{clientes}) = \text{sumatoriaCantidadAcciones}(\text{dameTítulos}(\text{claveCliente}, w.\text{clientes})))$

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`.  
 $(\forall \text{claveCliente} : \text{cliente}) (\text{def?}(\text{claveCliente}, w.\text{clientes}) \Rightarrow (\forall \text{nomTit} : \text{titulo}) (\text{def?}(\text{nomTit}, \text{dameTítulos}(\text{claveCliente}, w.\text{clientes})) \Rightarrow \text{cantidadPrometidasVenta}(\text{obtener}(\text{nomTit}, \text{dameTítulos}(\text{claveCliente}, w.\text{clientes})) \leq \text{cantidadAcciones}(\text{obtener}(\text{nomTit}, \text{dameTítulos}(\text{claveCliente}, w.\text{clientes}))))$

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

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

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

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

$\text{Rep} : \text{wolfie} \rightarrow \text{bool}$

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

$\text{sumatoriaAccionesTítulo} : t \text{ string} \times \text{clientes dict}(\text{cliente}, \text{infoCliente}) \rightarrow \text{nat}$

$\text{sumatoriaAccionesTítulo}(t, \text{clientes}) \equiv \text{sumatoriaAccionesTítuloConj}(t, \text{claves}(\text{clientes}), \text{clientes})$

$\text{sumatoriaAccionesTítuloConj} : t \text{ string} \times \text{claves conj}(\text{string}) \times \text{clientes dict}(\text{nat}, \text{infoCliente}) \rightarrow \text{nat}$

$\text{sumatoriaAccionesTítuloConj}(t, \text{claves}, \text{clientes}) \equiv \text{if } \emptyset?(\text{claves}) \text{ then } 0 \text{ else } \text{dameCantAcciones}(t, \text{dameTítulos}(\text{obtener}(\text{dameUno}(\text{claves}), \text{clientes}))) + \text{sumatoriaAccionesTítuloConj}(t, \text{sinUno}(\text{claves}), \text{clientes}) \text{ fi}$



```

cantidadDeCompra(promesas, nomTit)  $\equiv$  if vacia?(promesas) then
    0
else
    (if prim(promesas).nomTit = nomTit  $\wedge$  prim(promesas).tipo = compra then
        1
    else
        0
    fi) + cantidadDeCompra(fin(promesas), nomTit)
fi

```

cantidadDeVenta : *promesas secu(promesaTitulo)  $\times$  nomTit string  $\rightarrow$  nat*

```

cantidadDeVenta(promesas, nomTit)  $\equiv$  if vacia?(promesas) then
    0
else
    (if prim(promesas).nomTit = nomTit  $\wedge$  prim(promesas).tipo = venta then
        1
    else
        0
    fi) + cantidadDeVenta(fin(promesas), nomTit)
fi

```

dameTítulosCliente : *clientes dict(nat  $\times$  infoCliente)  $\times$  c cliente  $\rightarrow$  dictTítulos*

dameTítulosCliente(*clientes*, *c*)  $\equiv$  obtener(*clientes*, *c*).títulos

promesasALista : *títulos dict(string  $\times$  infoTituloCliente)  $\rightarrow$  secu(promesaTitulo)*

promesasALista(*títulos*)  $\equiv$  promesasAListaConClaves(claves(*títulos*), *títulos*)

promesasAListaConClaves : *claves conj(string)  $\times$  títulos dict(string  $\times$  infoTituloCliente)  $\rightarrow$  secu(promesaTitulo)*

```

promesasAListaConClaves(claves, títulos)  $\equiv$  if  $\emptyset?$ (claves) then
     $\langle \rangle$ 
else
    generarPromesasTitulo(dameUno(claves), obtener(títulos,
    dameUno(claves)) & promesasAListaConClaves(sinUno(claves),
    títulos)
fi

```

generarPromesasTitulo : *nomTit string  $\times$  info infoTituloCliente  $\rightarrow$  secu(promesaTitulo)*

```

generarPromesasTitulo(nomTit, info)  $\equiv$  (if info.promesas.compra.pendiente then
    generarPromesaTitulo(nomTit, compra, info.promesas.compra)
else
     $\langle \rangle$ 
fi) & (if info.promesas.venta.pendiente then
    generarPromesaTitulo(nomTit, venta, info.promesas.venta)
else
     $\langle \rangle$ 
fi)

```

generarPromesaTitulo : *nomTit string  $\times$  tipo string  $\times$  p promesa  $\rightarrow$  promesaTitulo*

generarPromesaTitulo(*nomTit*, *tipo*, *p*)  $\equiv$   $\langle$ *nomTit*, *tipo*, *p.umbral*, *p.cantidad* $\rangle$

## Función de abstracción



Abs :  $\text{wolfie } e \longrightarrow \text{wolfie} \quad \{\text{Rep}(e)\}$   
 Abs( $e$ )  $\equiv$  clientes( $w$ ) = claves( $\text{we.cientes}$ )  $\wedge$  títulos( $w$ ) = claves( $\text{we.titulos}$ )  $\wedge$  ( $\forall c : \text{cliente}, t : \text{titulo}$ ) ( $c \in \text{clientes}(w)$   $\wedge$   $t \in \text{títulos}(w) \Rightarrow \text{accionesPorCliente}(c, \text{nombre}(t), w) = \text{dameCantAcciones}(\text{nombre}(t), \text{dameTítulos}(c, \text{we.cientes}))$ )  $\wedge$  ( $\forall c : \text{cliente}$ ) ( $c \in \text{clientes}(w) \Rightarrow (\forall p : \text{promesa})$  ( $p \in \text{promesasDe}(c, w) \Leftrightarrow (\exists pEstr : \text{promesaTitulo} / pEstr \in \text{promesasAConj}(c, \text{we.cientes}) \wedge tp.tipo = \text{tipo}(p) \wedge tp.umbral = \text{limite}(p) \wedge tp.cantidad = \text{cantidad}(p) \wedge tp.nomTit = \text{titulo}(p))$ ))  
 promesasAConj :  $c \text{ cliente} \times \text{clientes } dict(\text{nat}, \text{infoCliente}) \longrightarrow \text{conj}(\text{tPromesa})$   
 promesasAConj( $c, \text{clientes}$ )  $\equiv$  damePromesas(obtener( $\text{clientes}, c$ ).titulos)  
 damePromesas : títulos  $dict(\text{string} \times \text{infoTituloCliente}) \longrightarrow \text{conj}(\text{promesaTitulo})$   
 damePromesas( $\text{titulos}$ )  $\equiv$  promesasAConjConClaves(claves( $\text{titulos}$ ),  $\text{titulos}$ )  
 promesasAConjConClaves : claves  $conj(\text{string}) \times \text{títulos } dict(\text{string} \times \text{infoTituloCliente}) \longrightarrow \text{secu}(\text{promesaTitulo})$   
 promesasAConjConClaves( $\text{claves}, \text{titulos}$ )  $\equiv$  **if**  $\emptyset?(\text{claves})$  **then**  
      $\emptyset$   
   **else**  
     generarPromesasTitulo(dameUno( $\text{claves}$ ), obtener( $\text{titulos}$ , dameUno( $\text{claves}$ ))  $\cup$  promesasAConjConClaves(sinUno( $\text{claves}$ ),  $\text{titulos}$ )  
   **fi**  
 generarPromesasTitulo : nomTit  $string \times \text{info } infoTituloCliente \longrightarrow \text{secu}(\text{promesaTitulo})$   
 generarPromesasTitulo( $\text{nomTit}, \text{info}$ )  $\equiv$  (**if**  $\text{info.promesas.compra.pendiente}$  **then**  
     generarPromesaTitulo( $\text{nomTit}, \text{compra}, \text{info.promesas.compra}$ )  
   **else**  
      $\emptyset$   
   **fi**)  $\cup$  (**if**  $\text{info.promesas.venta.pendiente}$  **then**  
     generarPromesaTitulo( $\text{nomTit}, \text{venta}, \text{info.promesas.venta}$ )  
   **else**  
      $\emptyset$   
   **fi**)  
 generarPromesaTitulo : nomTit  $string \times \text{tipo } string \times p \text{ promesa} \longrightarrow \text{promesaTitulo}$   
 generarPromesasTitulo( $\text{nomTit}, \text{tipo}, p$ )  $\equiv$   $\langle \text{nomTit}, \text{tipo}, p.umbral, p.cantidad \rangle$   
 dameCantAcciones : nomTit  $string \times \text{títulos } dict(\text{string}, \text{infoTituloCliente}) \longrightarrow \text{nat}$   
 dameCantAcciones( $\text{nomTit}, \text{titulos}$ )  $\equiv$  **if** def?( $\text{nomTit}, \text{titulos}$ ) **then**  $\prod_1(\text{obtener}(\text{nomTit}, \text{titulos}))$  **else** 0 **fi**  
 dameTítulos :  $c \text{ nat} \times \text{clientes } dict(\text{nat}, \text{infoCliente}) \longrightarrow dict(\text{string}, \text{infoTituloCliente}) \quad \{\text{def?}(c, \text{clientes})\}$   
 dameTítulos( $c, \text{clientes}$ )  $\equiv$   $\prod_1(\text{obtener}(c, \text{clientes}))$

## Algoritmos

```

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

iAgregarTítulo(in/out w: wolfie, in nomTit: string, in maxAcciones: nat, in cot: nat)
nat accionesDisponibles ← maxAcciones
bool enAlza ← true
nat rachaActual ← 0
nat rachaMaxima ← 0
nat fluctuaciones ← 0
tupla infoTítulo ← <maxAcciones, accionesDisponibles, cot, enAlza, rachaActual, rachaMaxima, fluctuaciones>
Definir(nomTit, infoTítulo, w.títulos);                // 0( |nomTit| )

```

```

iActualizarCotización(in/out w: wolfie, in nomTit: string, in cot: nat)
w.promesasDe.actualizado ← false
tupla infoTítulo ← Obtener(w.títulos, nomTit); // 0( |nomTit| )
if infoTítulo.cotización > cot then
  if infoTítulo.enAlza then
    infoTítulo.fluctuaciones ← infoTítulo.fluctuaciones + 1
  end
  infoTítulo.enAlza ← false infoTítulo.rachaActual ← 0
end
else
  if infoTítulo.enAlza then
    infoTítulo.fluctuaciones ← infoTítulo.fluctuaciones + 1
  end
  infoTítulo.enAlza ← true
  infoTítulo.rachaActual ← infoTítulo.rachaActual + 1
  if infoTítulo.rachaActual > infoTítulo.rachaMaxima then
    infoTítulo.rachaMaxima ← infoTítulo.rachaActual
  end
end
infoTítulo.cotización ← cot
// ejecutamos todas las promesas de venta
itDiccClientes itClientes ← crearIt(w.clientes)
while haySiguiente(itClientes) do
  ; // 0( #clientes )
  tupla infoCliente ← siguienteSignificado(itClientes)
  if definido(nomTit, infoCliente.títulos) then
    ; // 0( |nomTit| )
    tupla títuloActual ← obtener(infoCliente.títulos, nomTit); // 0( |nomTit| )
    nat accionesVendidas ← ejecutarVenta(títuloActual.promesas, cot); // 0( 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 arreglo de tuplas <cliente, totalAcciones> ordenado por la segunda
// componente de las tuplas
itDiccClientes itClientes ← crearIt(w.clientes); // 0( 1 )
nat cantidadClientes ← #claves(w.clientes)
arr clientesPorAcciones ← crearArreglo(cantidadClientes); // 0( #clientes )
nat i ← 0
while haySiguiente(itClientes) do
  ; // 0( #clientes )
  tupla clienteTotalAcciones ← <siguienteClave(itClientes), siguienteSignificado(itClientes).totalAcciones>
end
clientesPorAcciones ← mergeSort(clientesPorAcciones); // 0( 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

```

```

iEjecutarVenta(in/out promesas: promesas, in cot: nat)  $\longrightarrow$  res:nat
if promesas.venta.pendiente  $\wedge_L$  promesas.venta.umbral < cot then
  promesas.venta.pendiente  $\leftarrow$  false
  res  $\leftarrow$  promesas.venta.cantidad
else
  res  $\leftarrow$  0
end
end

iEjecutarCompra(in/out promesas: promesas, in cot: nat)  $\longrightarrow$  res:nat
if promesas.compra.pendiente  $\wedge_L$  promesas.compra.umbral < cot then
  promesas.compra.pendiente  $\leftarrow$  false
  res  $\leftarrow$  promesas.compra.cantidad
else
  res  $\leftarrow$  0
end
end

iAgregarPromesa(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  $\leftarrow$  false
end
tupla infoCliente  $\leftarrow$  obtener(c, w.clientes)
if definido(infoCliente.titulos, nomTit) then
  tupla infoTituloCliente  $\leftarrow$  obtener(infoCliente.titulos, nomTit)
  if tipo = venta then
    infoTituloCliente.promesas.venta  $\leftarrow$  <true, umbral, cantidad>
  end
  if tipo = compra then
    infoTituloCliente.promesas.compra  $\leftarrow$  <true, umbral, cantidad>
  end
end
else
  tupla infoTituloCliente
  infoTituloCliente.cantidadAcciones  $\leftarrow$  0
  if tipo = venta then
    infoTituloCliente.promesas.compra  $\leftarrow$  <false, 0, 0>
    infoTituloCliente.promesas.venta  $\leftarrow$  <true, umbral, cantidad>
  end
  if tipo = compra then
    infoTituloCliente.promesas.venta  $\leftarrow$  <false, 0, 0>
    infoTituloCliente.promesas.compra  $\leftarrow$  <true, umbral, cantidad>
  end
  definir(infoCliente.titulos, nomTit, infoTituloCliente)
end

iclientes()  $\longrightarrow$  res: itDiccClientes(nat)
res  $\leftarrow$  crearItDiccOrd(w.clientes)

ititulos()  $\longrightarrow$  res: itDiccTitulos(nat)
res  $\leftarrow$  crearItDiccTitulos(w.titulos)

```

```

iPromesasDe(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  $\leftarrow$  vacía()
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)

iAccionesPorCliente(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

iEnAlza(in w: wolfie, in nomTit: string)  $\longrightarrow$  res: bool
tupla infoTítulo  $\leftarrow$  obtener(w.títulos, nomTit)
res  $\leftarrow$  infoTítulo.enAlza)

imaximaRacha(in w: wolfie, in nomTit: string)  $\longrightarrow$  res: nat
tupla infoTítulo  $\leftarrow$  obtener(w.títulos, nomTit)
res  $\leftarrow$  infoTítulo.maximaRacha

itituloMasVolatil(in w: wolfie, in nomTit: string)  $\longrightarrow$  res: nomTit
itTítulos  $\leftarrow$  crearIt(w.títulos)
nomTitMaximo  $\leftarrow$  siguiente(itTítulos)
maxFluctuacion  $\leftarrow$  Obtener(nomTitMaximo, w.títulos).fluctuaciones
avanzar(itTítulos)
while haySiguiente?(itTítulos) do
  flucActual  $\leftarrow$  Obtener(siguiente(itTítulos), w.títulos).fluctuaciones
  if flucActual > maxFluctuacion then
    maxFluctuacion  $\leftarrow$  flucActual
    nomTitMaximo  $\leftarrow$  siguiente(itTítulos)
  end
end
res  $\leftarrow$  nomTitMaximo

```

### 3. Módulo Diccionario Títulos

#### Interfaz

**parámetros formales**

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

**se explica con:**  $\text{DICCIONARIO}(\text{STRING}, \alpha)$ .

**géneros:**  $\text{diccTítulo}(\text{String}, \alpha)$ ,  $\text{itDicc}(\alpha)$ .

**Operaciones básicas de diccionario títulos**

$\text{NUEVO DICCIONARIO}() \rightarrow \text{res} : \text{diccTítulo}(\text{String}, \alpha)$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{res} =_{\text{obs}} \text{vacío}()\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Crea un nuevo diccionario vacío.

$\text{DEFINIR}(\text{in } c : \text{string}, \text{in } s : \alpha, \text{in/out } d : \text{diccTítulo}(\text{String}, \alpha))$

**Pre**  $\equiv \{d =_{\text{obs}} d_0\}$

**Post**  $\equiv \{d =_{\text{obs}} \text{definir}(d_0, c, s)\}$

**Complejidad:**  $\Theta(|c| + \text{copy}(s))$

**Descripción:** Define la clave  $c$  con el significado  $s$  en el diccionario  $d$ .

**Aliasing:** Se agrega por copia el significado  $s$

$\text{OBTENER}(\text{in } c : \text{string}, \text{in } d : \text{diccTítulo}(\text{String}, \alpha)) \rightarrow \text{res} : \alpha$

**Pre**  $\equiv \{\text{def?}(c, d)\}$

**Post**  $\equiv \{\text{res} =_{\text{obs}} \text{obtener}(c, d)\}$

**Complejidad:**  $\Theta(|c|)$

**Descripción:** Devuelve el significado de la clave  $c$  contenida en el diccionario  $d$ .

$\text{DEFINIDO?}(\text{in } c : \text{string}, \text{in } d : \text{diccTítulo}(\text{String}, \alpha)) \rightarrow \text{res} : \text{bool}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{res} =_{\text{obs}} \text{def?}(c, d)\}$

**Complejidad:**  $\Theta(|c|)$

**Descripción:** Chequea si está definida la clave  $c$  en el diccionario  $d$ .

$\#\text{CLAVES}(\text{in } d : \text{diccTítulo}(\text{String}, \alpha)) \rightarrow \text{res} : \text{nat}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{res} =_{\text{obs}} \#(\text{claves}(d))\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve la cantidad de claves del diccionario.

**Operaciones del iterador**

$\text{CREARIT}(\text{in } d : \text{diccTítulo}(\text{String}, \alpha)) \rightarrow \text{res} : \text{itDicc}(\text{String}, \alpha)$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{res} =_{\text{obs}} \langle \text{crearItUni}(\langle \rangle, d.\text{claves}), \text{crearItUni}(\langle \rangle, d.\text{alfas}) \rangle\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Crea un iterador unidireccional del diccionario. Se pueden recorrer los elementos aplicando iterativamente Siguiente

$\text{HAYSIGUIENTE?}(\text{in } it : \text{itDicc}(\text{String}, \alpha)) \rightarrow \text{res} : \text{bool}$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{res} =_{\text{obs}} \text{haySiguiente?}(it)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve true si y sólo si en el iterador todavía quedan elementos para avanzar

$\text{SIGUIENTE}(\text{in } it : \text{itDicc}(\text{String}, \alpha)) \rightarrow \text{res} : \text{tupla}(\text{String}, \alpha)$

**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  $\alpha$  y es modificable si y sólo si  $it$  es modificable. En cambio,  $res.clave$  no es modificable

**SIGUIENTEC clave**(**in**  $it : \text{itDicc}(String, \alpha) \rightarrow res : String$

**Pre**  $\equiv \{\text{HaySiguiente?}(it)\}$

**Post**  $\equiv \{\text{alias}(res =_{\text{obs}} \text{Siguiente}(it).clave)\}$

**Complejidad:**  $\Theta(1)$

**Descripción:** Devuelve la clave del elemento siguiente del iterador

**Aliasing:**  $res$  no es modificable

**SIGUIENTESIGNIFICADO**(**in**  $it : \text{itDicc}(String, \alpha) \rightarrow res : \alpha$

**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**  $it : \text{itDicc}(String, \alpha)$

**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:

$\text{diccTítulo}(String, \alpha)$  se representa con  $\text{estr\_diccTítulo}$

donde  $\text{estr\_diccTítulo}$  es  $\text{tupla}(\text{raíz} : \text{estr\_nodo}, \text{claves} : \text{lista}(String))$

donde  $\text{estr\_nodo}$  es  $\text{tupla}(\text{significado} : \text{puntero}(\alpha), \text{hijos} : \text{array}[256] \text{ de } \text{puntero}(\text{estr\_nodo}))$

### Invariante de representación:

1. Dos nodos no pueden compartir un hijo
2. Sin ciclos en el árbol
3. Las hojas del árbol no pueden tener significado nulo
4. La cantidad de claves ingresadas en  $e.claves$  debe ser igual a la cantidad de significados válidos (distintos de NULL) del árbol Trie
5. Las claves contenidas en  $e.claves$  deben estar definidas en el árbol Trie

$\text{Rep} : \text{estr\_diccTítulo} \rightarrow \text{bool}$

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

$\text{SinCiclos}(e.raíz, \emptyset) \wedge_L$

$\text{NoCompartenHijos}(e.raíz) \wedge_L$

$\text{SignificadosHojasNotNull}(e.raíz) \wedge$

$\text{Longitud}(e.claves) =_{\text{obs}} \text{CantSignificados}(e.raíz) \wedge$

$(\forall \text{clave} : String) ((\text{está?}(\text{clave}, e.claves)) \iff \text{definido?}(\text{clave}, e.raíz))$

1.  
1.  
2.  
3.  
4.

$\text{SignificadoHojaNotNull} : \text{estr\_nodo } e \rightarrow \text{bool}$

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

Hoja? :  $\text{estr\_nodo } e \times \text{nat} \rightarrow \text{bool}$   
 Hoja?( $e, n$ )  $\equiv (e.\text{hijos}[n] =_{\text{obs}} \text{NULL}) \wedge \text{if } (n < 256) \text{ then Hoja?}(e, n+1) \text{ else true fi}$

RecorrerHijos :  $\text{estr\_nodo } e \times \text{nat} \rightarrow \text{bool}$  {SinCiclos( $e, 0$ )}  
 RecorrerHijos( $e, n$ )  $\equiv (\text{if } (\neg(e.\text{hijos}[n] =_{\text{obs}} \text{NULL})) \text{ then SignificadoHojaNotNull}(*e.\text{hijos}[n]) \text{ else true fi}) \wedge$   
 $(\text{if } (n < 256) \text{ then RecorrerHijos}(e, n+1) \text{ else true fi}) \text{ FI}$

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

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

**Función de abstracción:**

Abs :  $\text{estr\_diccTítulo } e \rightarrow \text{dicc}(\text{string}, \alpha)$  {Rep( $e$ )}  
 Abs( $e$ )  $=_{\text{obs}} d: \text{dicc}(\text{string}, \alpha) \mid \#(\text{claves}(d)) =_{\text{obs}} \text{Longitud}(e.\text{clave}) \wedge_L$  1.  
 $(\forall c: \text{string})(\text{def?}(c, d) \Rightarrow_L$  2.  
 $(\text{definido?}(c, e.\text{raíz}) \wedge_L$   
 $\text{obtener}(c, d) =_{\text{obs}} *(\text{ObtDeEstruc}(c, e.\text{raíz})) ) )$

definido? :  $\text{string } e \times \text{estr\_nodo } e \rightarrow \text{bool}$   
 definido?( $c, n$ )  $\equiv \text{if vacía?}(c) \text{ then}$   
 $\neg(n.\text{significado} = \text{NULL})$   
 $\text{else}$   
 $\text{if } n.\text{hijos}[\text{ORD}(\text{prim}(c))] = \text{NULL} \text{ then false else definido?}(\text{fin}(c), n.\text{hijos}[\text{ORD}(\text{prim}(c))]) \text{ fi}$   
 $\text{fi}$

ObtDeEstruc :  $\text{string } e \times \text{estr\_nodo } e \rightarrow \alpha$   
 ObtDeEstruc( $c, n$ )  $\equiv \text{if vacía?}(c) \text{ then } n.\text{significado} \text{ else ObtDeEstruc}(\text{fin}(c), n.\text{hijos}[\text{ORD}(\text{prim}(c))]) \text{ fi}$

**Representación del iterador:**

El iterador del diccionario es simplemente un iterador a la lista de claves.

Lo único que hay que pedir es que satisfaga el Rep de esta lista.

Por implementación, alcanza con que sea unidireccional.

$\text{itDiccTítulos}(\text{String}, \alpha)$  se representa con  $\text{itDic}$

donde  $\text{itDic}$  es  $\text{tupla}(\text{claves: itLista}(\text{String}) )$

Rep :  $\text{itDic} \rightarrow \text{bool}$

Rep( $it$ )  $\equiv \text{true} \iff$   
 $\text{Rep}(it.\text{claves})$

Abs :  $\text{itDic } it \rightarrow \text{itUni}(\text{String})$  {Rep( $it$ )}

Abs( $it$ )  $\equiv \text{CrearItUni}(\text{Siguietes}(it.\text{claves}))$

**Algoritmos**

$i\text{NuevoDiccionario}() \rightarrow \text{res: bool}$

**Orden Complejidad:**  $O(1)$

$\text{res} \leftarrow \langle \text{raíz: } i\text{NuevoNodo}(), \text{Claves: Vacía}(), \text{Alfas: Vacía}() \rangle;$  // 0(1)

$i\text{NuevoNodo}() \rightarrow \text{res: estr\_nodo}$

**Orden Complejidad:**  $O(1)$



```

res ← <significado: NULL, hijos: CrearArreglo() >; // 0(1)
for var i : nat ← 0 to 255; // 0(256)
do
    res.hijos[i] ← NULL
end

iDefinir(in c: string, in s: α, in/out d: estr_diccTítulo) → res: estr_dicTrie
Orden Complejidad:  $O(|c|)$ 
var actual : puntero(estr_nodo) ← &(d.raíz); // 0( 1 )
for var i : nat ← 0 to (Longitud(c) - 1); // 0(|c|)
do
    if actual → hijos[ORD(c[i])] = NULL then // 0(|c|)
        AgregarAtrás(d.Claves, c);
        actual → hijos[ORD(c[i])] ← &(iNodoNuevo())
    end
    actual ← (actual → hijos[ORD(c[i])])
end
actual → significado ← &(Copiar(s)); // 0( 1 )

iObtener(in c: string, in d: estr_diccTítulo) → res: α
Orden Complejidad:  $O(|c|)$ 
var actual : puntero(estr_nodo) ← &(d.raíz); // 0( 1 )
for var i : nat ← 0 to (Longitud(c) - 1); // 0(|c|)
do
    actual ← (actual → hijos[ORD(c[i])])
end
res ← * (actual → significado); // 0( 1 )

iDefinido?(in c: string, in d: estr_diccTítulo) → res: bool
Orden Complejidad:  $O(|c|)$ 
var actual : puntero(estr_nodo) ← &(d.raíz); // 0( 1 )
var i : Nat ← 0; // 0( 1 )
while actual != NULL && (i < Longitud(c)); // 0(|c|)
do
    i ← i + 1
    actual ← (actual → hijos[ORD(c[i])])
end
if (actual != NULL) then
    if (actual → significado) != NULL then
        res ← true; // 0( 1 )
    else
        res ← false; // 0( 1 )
    end
else
    res ← false; // 0( 1 )
end

i#Claves(in d: estr_diccTítulo) → res: nat
Orden Complejidad:  $O(1)$ 
res ← Longitud(d.Claves); // 0( 1 )

```

### Algoritmos del iterador

```

iCrearIT(in d: diccTítulo(String, α)) → res: ItDicc(String, α)
Orden Complejidad:  $O(1)$ 
res ← claves: CrearIt(d.claves)

```

*i*HaySiguiente?(in it: ItDicc(*String*, $\alpha$ ))  $\rightarrow$  res: bool

**Orden Complejidad:**  $O(1)$

res  $\leftarrow$  HaySiguiente(it.claves)

*i*Siguiente(in it: ItDicc(*String*, $\alpha$ ))  $\rightarrow$  res: *String*

**Orden Complejidad:**  $O(1)$

res  $\leftarrow$  Siguiente(it.claves)

*i*Avanzar(in it: ItDicc(*String*, $\alpha$ ))  $\rightarrow$  res: ItDicc(*String*)

**Orden Complejidad:**  $O(1)$

res  $\leftarrow$  Avanzar(it.claves)

## 4. Módulo Diccionario Clientes

### Interfaz

#### parámetros formales

**géneros**  $\alpha$

**función** COPIAR(in  $a: \alpha$ )  $\rightarrow$  res :  $\alpha$

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} a\}$

**Complejidad:**  $\Theta(\text{copy}(a))$

**Descripción:** función de copia de  $\alpha$ 's

**se explica con:** DICCIONARIO( $\alpha, \sigma$ ), ITERADOR UNIDIRECCIONAL(TUPLA( $\alpha, \sigma$ )).

**géneros:** dictClientes(nat, infoCliente), itDictClientes(Tupla( $\alpha, \sigma$ )).

#### Operaciones básicas de diccionario ordenado

VACÍO(in  $n: \text{nat}$ )  $\rightarrow$  res : dictClientes(nat, infoCliente)

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{res =_{\text{obs}} \text{vacío}\}$

**Complejidad:**  $O(n)$

**Descripción:** genera un diccionario vacío.

DEFINIR(in/out  $d: \text{dict}(\text{nat}, \text{infoCliente})$ , in  $c: \text{nat}$ , in  $s: \text{infoCliente}$ )

**Pre**  $\equiv \{d =_{\text{obs}} d_0\}$

**Post**  $\equiv \{d =_{\text{obs}} \text{definir}(d_0, c, s)\}$

**Complejidad:**  $O(\#claves(d) + \text{copy}(c) + \text{copy}(s))$

**Descripción:** define la clave  $c$  con el significado  $s$  en el diccionario.

**Aliasing:** los elementos  $c$  y  $s$  se definen por copia.

OBTENER(in  $d: \text{dictClientes}(\text{nat}, \text{infoCliente})$ , in  $c: \text{nat}$ )  $\rightarrow$  res : infoCliente

**Pre**  $\equiv \{\text{def?}(c, d)\}$

**Post**  $\equiv \{\text{esAlias}(res, \text{obtener}(c, d))\}$

**Complejidad:**  $O(\log(n))$

**Descripción:** obtiene el significado  $\sigma$  que corresponde a la clave  $c$ .

**Aliasing:** se genera aliasing entre  $res$  y el significado  $\sigma$

#### Operaciones del iterador

CREARIT(in  $d: \text{dicc}(\text{nat}, \text{infoCliente})$ )  $\rightarrow$  res : itDicc( $\alpha, \sigma$ )

**Pre**  $\equiv \{\text{true}\}$

**Post**  $\equiv \{\text{alias}(\text{esPermutacion}(\text{SecuSuby}(res), d)) \wedge \text{vacía?}(\text{Anteriores}(res))\}$

**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* =<sub>obs</sub> 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( $\alpha, \sigma$ )

**Pre**  $\equiv$  {HaySiguiente?(*it*)}

**Post**  $\equiv$  {alias(*res* =<sub>obs</sub> 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.

SIGUIENTECLOVE(**in** *it*: itDicc(nat, infoCliente))  $\rightarrow$  *res* : nat

**Pre**  $\equiv$  {HaySiguiente?(*it*)}

**Post**  $\equiv$  {alias(*res* =<sub>obs</sub> 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* =<sub>obs</sub> 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* =<sub>obs</sub> 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), *tamano*: nat)

*Invariante de representación*

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

Rep : *dc*  $\rightarrow$  bool

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



```

idefinir(in/out dc: dictClientes(nat, infoCliente), in c: nat, in s: infoCliente )
posActual ← dc.tamano
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.tamano += 1

iobtener(in/out dc: dictClientes(nat, infoCliente), in c: nat ) → res: infoCliente
der ← dc.tamano-1
izq ← 0
medio ← dc.tamano/2
while dc.clave[medio] != c do
    if dc.clave[medio] > c then
        der ← medio
    end
    if dc.clave[medio] < c then
        izq ← medio
    end
end
res ← dc.significados[medio]

icrearIt(in/out dc: dictClientes(nat, infoCliente)) → res: itDictClientes(nat, infoCliente)
it.posicion ← 0
it.limite ← tamano(dc.significados)
it.claves ← &(dc.claves)
it.significados ← &(dc.significados)
res ← it

ihaySiguiente(in it: itDictClientes(nat, infoCliente)) → res: bool
res ← it.posición < it.límite

isiguiente(in/out dc: itDictClientes(nat, infoCliente)) → res: tupla(nat, infoCliente)
res ← <*(it.claves[it.posición]), *(it.significados[it.posición])>

isiguienteClave(in/out dc: itDictClientes(nat, infoCliente)) → res: nat
res ← *(it.claves[it.posición])

isiguienteSignificado(in/out dc: itDictClientes(nat, infoCliente)) → res: infoCliente
res ← *(it.significados[it.posición])

iavanzar(in/out dc: itDictClientes(nat, infoCliente))
it.posición ← it.posición + 1

```