DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Practico 2

Algoritmos y Estructura de Datos II
Primer cuatrimestre 2014

Grupo 10

Integrante	LU	Correo electrónico
Gómez, Pablo Nicolás	156/13	mago-1986@hotmail.com
Parral, Lucía Inés	162/13	luciaparral@gmail.com
Roulet, Nicolás	587/13	nicoroulet@gmail.com
Tamborindeguy, Guido Joaquín	584/13	guido@tamborindeguy.com.ar

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) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En los distintos módulos se asume que los tipos básicos se pasan por copia, con complejidad O(1). El tipo STRING se pasa también por copia, y esta operación tiene complejidad O(n), con n la longitud del string. Para los módulos diseñados, el pasaje es por referencia. De todos modos, las aclaraciones que nos parecieron necesarias se encuentran en la sección de aliasing de la interfaz.

1.1. Renombres de Módulos

Módulo Dinero es Nat
Módulo Cliente es Nat
Módulo TipoPromesa es enum{compra, venta}
Módulo Nombre es String

2. Módulo Wolfie

2.1. Interfaz

géneros wolfie, itTítulos
se explica con: WOLFIE, ITERADOR UNIDIRECCIONAL

2.1.1. Operaciones básicas de wolfie

CLIENTES(**in** w: `wolfie`) → res : `itConjEstNat(cliente)`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ crearItUni(clientes(w))}
Complejidad: $O(1)$
Descripcion: Devuelve un iterador a los clientes de un wolfie.

TÍTULOS(**in** w: `wolfie`) → res : `itUni(título)`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ crearItUni(títulos(w))}
Complejidad: $O(1)$
Descripcion: Devuelve un iterador a los títulos de un wolfie.

PROMESASDE(**in** c: `cliente`, **in** w: `wolfie`) → res : `itConj(promesa)`
Pre ≡ {$c \in$ clientes(w)}
Post ≡ {$res =_{\text{obs}}$ crearItUni(promesasDe(c, w))}
Complejidad: $O(T \cdot C \cdot |max\_nt|)$
Descripcion: Devuelve un iterador a las promesas de un wolfie

ACCIONESPORCLIENTE(**in** c: `cliente`, **in** nt: `nombre`, **in** w: `wolfie`) → res : nat
Pre ≡ {$c \in$ clientes(w) ∧ (∃ t:título) ($t \in$ títulos(w) ∧ nombre(t) = nt)}
Post ≡ {$res =_{\text{obs}}$ accionesPorCliente(c, nt, w)}
Complejidad: $O(log(C) + |nt|)$
Descripcion: Devuelve la cantidad de acciones que un cliente posee de un determinado título.

INAUGURARWOLFIE(**in** cs: `conj(cliente)`) → res : `wolfie`
Pre ≡ {$\neg\emptyset?(cs)$}
Post ≡ {$res =_{\text{obs}}$ inaugurarWolfie(cs)}
Complejidad: $O(\#(cs)^2)$
Descripcion: Crea un nuevo wolfie a partir de un conjunto de clientes.

AGREGARTÍTULO(**in** t: `título`, **in/out** w: `wolfie`)
Pre ≡ {$w_0 =_{\text{obs}} w$ ∧ (∀ $t2$:título) ($t2 \in$ títulos(w) ⇒ nombre(t) ≠ nombre($t2$)}

Post $\equiv \{w =_{\text{obs}} \text{agregarTítulo}(t, w_0\}$
Complejidad: $O(|nombre(t)| + C)$

ActualizarCotización(**in** nt : `nombre`, **in** cot : `nat`, **in/out** w : `wolfie`)
Pre $\equiv \{w_0 =_{\text{obs}} w \wedge (\exists\ t$:título) $(t \in \text{títulos}(w) \wedge \text{nombre}(t) = nt)\}$
Post $\equiv \{w =_{\text{obs}} \text{actualizarCotización}(nt, cot, w_0)\}$
Complejidad: $O(C \cdot |nt| + C \cdot log(C))$
Descripcion: Cambia la cotización de un determinado título. Esta operación genera que se desencadene el cumplimiento de promesas (según corresponda): primero de venta y luego, de compra, según el orden descendente de cantidad de acciones por título de cada cliente.

AgregarPromesa(**in** c : `cliente`, **in** p : `promesa`, **in/out** w : `wolfie`)
Pre $\equiv \{w_0 =_{\text{obs}} w \wedge (\exists t$: título) $(t \in \text{títulos}(w) \wedge \text{nombre}(t) = \text{título}(p)) \wedge c \in \text{clientes}(w) \wedge_{\text{L}} (\forall p2$: promesa) $(p2 \in \text{promesasDe}(c, w) \Rightarrow (\text{título}(p) \neq \text{título}(p2) \vee \text{tipo}(p) \neq \text{tipo}(p2))\ ) \wedge (\text{tipo}(p) = \text{vender} \Rightarrow \text{accionesPorCliente}(c, \text{título}(p), w) \geq \text{cantidad}(p)))\}$
Post $\equiv \{w =_{\text{obs}} \text{agregarPromesa}(c, p, w_0)\}$
Complejidad: $O(|título(p)| + log(C))$
Descripcion: Agrega una nueva promesa.

EnAlza(**in** nt : `nombreTítulo`, **in** w : `wolfie`) $\rightarrow$ res : `bool`
Pre $\equiv \{(\exists t$: título) $(t \in \text{títulos}(w) \wedge \text{nombre}(t) = \text{nt})\}$
Post $\equiv \{res =_{\text{obs}} \text{enAlza}(nt, w)\}$
Complejidad: $O(|nt|)$
Descripcion: Dado un título, informa si está o no en alza.

2.1.2. Operaciones básicas de itTítulos

CrearIt(**in** w : `wolfie`) $\rightarrow$ res : `itTítulos`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{crearItUni(titulos(w))}\}$
Complejidad: $O(1)$
Descripcion: Devuelve un iterador unidireccional a los títulos de wolfie.

Actual(**in** i : `itTítulos`) $\rightarrow$ res : `título`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{actual(i)}\}$
Complejidad: $O(|titulo(actual(i))|)$
Descripcion: Devuelve el título actual.

Próximo(**in/out** i : `itTítulos`) $\rightarrow$ res : `[`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{\text{H}\}\text{ayPróximo(i)} \wedge i_0 = i]\ i =_{\text{obs}} \text{avanzar}(i_0)$ $[O(1)]$ [Avanza el iterador.]

HayPróximo(**in/out** i : `itTítulos`) $\rightarrow$ res : `bool`
Pre $\equiv \{\text{true}\}$
Post $\equiv \{res =_{\text{obs}} \text{hayMas(i)}\}$
Complejidad: $O(1)$
Descripcion: Pregunta si hay más elementos para iterar.

2.2. Representación

2.2.1. Representación de wolfie

`wolfie` se representa con `estr`

donde `estr` es `tupla`(*títulos*: `diccString(infoTítulo)`,
clientes: `conjEstNat(cliente)`
últimoLlamado: `<`*cliente*`: cliente,` *promesas*`:conj(promesa),` *fueUltimo*`: bool>`)

donde `infoTítulo` es `tupla`(*arrayClientes*: `array_dimensionable(tuplaPorCliente),` *título*: `título,` *accDisponibles*: `nat`)

donde `tuplaPorCliente` es `tupla`(*cliente*: `cliente,` *cantAcc*: `nat,` *promCompra*: `*promesa,` *promVenta*: `*promesa`)
Con un orden definido por $a<b \Leftrightarrow a.cliente < b.cliente$

donde `tuplaPorCantAcc` es `tupla`(*cliente*: `cliente,` *cantAcc*: `nat,` *promCompra*: `*promesa,` *promVenta*: `*promesa`)
Con un orden definido por $a<b \Leftrightarrow a.cantAcc > b.cantAcc$

2.2.2. Invariante de representación de wolfie

(I) Los clientes de *clientes* son los mismos que hay dentro de *titulos*.

(II) Las promesas de compra son de su título y cliente y no cumplen los requisitos para ejecutarse.

(III) Las promesas de venta son de su título y cliente y no cumplen los requisitos para ejecutarse.

(IV) Las acciones disponibles de cada título son el máximo de acciones de ese título menos la suma de las acciones de ese titulo que tengan los clientes, y son mayores o iguales a 0.

(V) El *cliente* de *últimoLlamado* pertenece a *clientes*.

(VI) En *últimoLlamado*, si $fueÚltimo$ es true, las promesas de *promesas* son todas las promesas que tiene *cliente*.

(VII) Los clientes están ordenados en *arrayClientes* de *e.titulos*.

(VIII) Los títulos en infoTítulo tienen el mismo nombre que la clave que lleva a ellos.

$$Rep : estr \longrightarrow bool$$

Rep(e) $\equiv$ true $\Longleftrightarrow$
(I)($\forall$ c: cliente) $\big($pertenece?(c, $e.clientes$) $\Leftrightarrow$ ($\exists$ t: título) $\big($def?(t, $e.titulos$) $\wedge_{\text{L}}$estáCliente?(c, Significado(t, $e.titulos$).$arrayClientes$)$\big)\big)$ $\wedge_{\text{L}}$

(II)($\forall$ p: $*$promesa, t: nombre, c: cliente) $\big($(p $\neq$ NULL $\wedge$ def?(t, $e.titulos$) $\wedge_{\text{L}}$estáCliente?(c, Significado(t, $e.titulos$).$arrayClientes$) $\wedge_{\text{L}}$buscarCliente(c, Significado(t, $e.titulos$).$arrayClientes$).$promCompra$=p) $\Rightarrow_{\text{L}}$título($*p$)=t $\wedge$ tipo($*p$)=compra $\wedge$ (límite($*p$)>cotización(Significado(t, $e.titulos$).$titulo$) $\vee$ cantidad($*p$)>Significado(t, $e.titulos$).$accDisponibles$)$\big)$ $\wedge$

(III)($\forall$ p: $*$promesa, t: nombre, c: cliente) $\big($(p $\neq$ NULL $\wedge$ def?(t, $e.titulos$) $\wedge_{\text{L}}$estáCliente?(c, Significado(t, $e.titulos$).$arrayClientes$) $\wedge_{\text{L}}$buscarCliente(c, Significado(t, $e.titulos$).$arrayClientes$).$promVenta$=p) $\Rightarrow_{\text{L}}$(título($*p$)=t $\wedge$ tipo($*p$)=venta $\wedge$ límite($*p$) <cotización(Significado(t, $e.titulos$).$titulo$)))$\big)$ $\wedge$

(IV)($\forall$ nt: nombreT) (def?(nt, $e.titulos$) $\Rightarrow_{\text{L}}$(Significado(nt, $e.titulos$).$accDisponibles$ = máximo(Significado(nt, $e.titulos$).$titulo$) - sumaAccClientes(Significado(nt, $e.titulos$).$arrayClientes$, 0) $\wedge$ Significado(nt, $e.titulos$).$accDisponibles$ $\geq$ 0)) $\wedge$

(V)(pertenece?($e.últimoLlamado.cliente$, $e.clientes$)) $\wedge_{\text{L}}$

(VI)($e.últimoLlamado.fueÚltimo$ $\Rightarrow$ ($\forall$ p: promesa) $\big($pertenece?(p, $e.últimoLlamado.promesas$) $\Leftrightarrow$ $\big($def?(título(p), $e.titulos$) $\wedge_{\text{L}}$
if tipo(p)=compra **then**
buscarCliente($e.últimoLlamado.cliente$, Significado(título(p), $e.titulos$).$arrayClientes$).promCompra = p
else
buscarCliente($e.últimoLlamado.cliente$, Significado(título(p), $e.titulos$).$arrayClientes$).promVenta = p
fi$\big)\big)$

(VII)($\forall$ t: nombre) def?(t, $e.titulos$) $\Rightarrow_{\text{L}}\big($($\forall$ i:nat) i<longitud(buscar(t, $e.titulos$).$arrayClientes$)-1 $\Rightarrow$ (buscar(t, $e.titulos$).$arrayClientes$)[i] <(buscar(t, $e.titulos$).$arrayClientes$)[i+1]$\big)$

(VIII)($\forall$ t: nombre) def?(t, $e.titulos$) $\Rightarrow_{\text{L}}t$ = nombre(Significado(t, $e.titulos$).$titulo$)

estáCliente? : cliente $\times$ array_dimensionable(tuplaPorCliente) $\longrightarrow$ bool
estáCliente?(c, a) $\equiv$ auxEstáCliente($c, a, 0$)

auxEstáCliente : cliente $\times$ array_dimensionable(tuplaPorCliente) $\times$ nat $\longrightarrow$ bool
auxEstáCliente(c, a, i) $\equiv$ **if** i=longitud(a) **then** false **else** $a[i].cliente = c \vee$ auxEstáCliente($c, a, i+1$) **fi**

buscarCliente : cliente $\times$ array_dimensionable(tuplaPorCliente) $\longrightarrow$ tuplaPorCliente {estáCliente(c, a)}
buscarCliente(c, a) $\equiv$ auxBuscarCliente($c, a, 0$)

auxBuscarCliente : cliente $\times$ array_dimensionable(tuplaPorCliente) $\times$ nat $\longrightarrow$ tuplaPorCliente {estáCliente(c, a)}
auxBuscarCliente(c, a, i) $\equiv$ **if** $a[i].cliente = c$ **then** $a[i]$ **else** auxBuscarCliente($c, a, i+1$) **fi**
sumaAccClientes : array_dimensionable(tuplaPorCliente) $\times$ nat $\longrightarrow$ nat
auxBuscarCliente(a, i) $\equiv$ **if** i=longitud(a) **then** 0 **else** $a[i].cantAcc$ + sumaAccClientes($a, i+1$) **fi**

2.2.3. Función de abstracción de wolfie

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

Abs(e) $=_{\text{obs}}$ w: wolfie | clientes(w)=$e.clientes$ $\wedge$ ($\forall t$: título) $\big($$t$ $\in$títulos(w)$\Leftrightarrow$(def?(t, $e.titulos$)$\wedge_{\text{L}}t$=Significado(t, $e.titulos$).$titulo$)$\big)$
$\wedge$ ($\forall$ c: cliente) promesasDe(c, w)=damePromesas(crearIt($e.titulos$), e, c) $\wedge$
accionesPorCliente(c, t, w)=buscarCliente(Significado(t, $e.titulos$).$arrayClientes$).$cantAcc$

damePromesas : itClaves(diccString) $\times$ estr $\times$ cliente $\longrightarrow$ conj(promesa)

damePromesas(it, e, c) $\equiv$ **if** hayMas?(it) **then**
if buscarCliente(Significado(actual(it), e.titulos)).$promCompra \neq$ NULL **then**
{buscarCliente(Significado(actual(it), e.titulos)).$promCompra \neq$ NULL} $\cup$ **fi**
if buscarCliente(Significado(actual(it)e.titulos)).$promVenta \neq$ NULL **then**
{buscarCliente(Significado(actual(it), e.titulos)).$promVenta \neq$ NULL} $\cup$ **fi**
damePromesas(avanzar(it), e, c)
else
vacio
fi

2.2.4. Representación de itTítulos

`itTítulos` **se representa con** `iterador`

donde `iterador` es `tupla`(it: `itClaves(infoTítulo)`,
$dicc$: `*diccString(infoTítulo)`)

2.2.5. Invariante de Representación de itTítulos

Los el iterador de claves es iterador del diccString.

Rep : iterador $\longrightarrow$ bool

Rep(i) $\equiv$ true $\iff$ esIterador(i.it, CrearIt(*(i.dicc)))

esIterador : itClaves(infoTítulo) $\times$ itClaves(infoTítulo) $\longrightarrow$ bool

esIterador(it1, it2 $\equiv$ actual($it1$)=actual($it2$) $\vee$ (hayMas?($it2$) $\wedge_{\text{L}}$esIterador($it1$, Avanzar(it2)))

2.2.6. Función de abstracción de itTítulos

Abs : iterador i $\longrightarrow$ itTítulos {Rep(i)}

Abs(i) $=_{\text{obs}}$ t: itTítulos | actual(t)=Significado(actual(i.it), *(i.dicc)) $\wedge$ (hayMás(i.it)$\Rightarrow_{\text{L}}$(hayMás(t) $\wedge_{\text{L}}$Abs(<Avanzar(i.it), i.dicc>, avanzar(t))))

2.3. Algoritmos

2.3.1. Algoritmos de wolfie

iClientes(in e: estr) $\rightarrow$ res: itConjEstNat

```
1  return(CrearIt(e.clientes))
```

Complejidad: O(1)

iTítulos(in e: estr) $\rightarrow$ res: itTítulos

```
1  return(CrearIt(e))
```

Complejidad: O(1)

iPromesasDe(in c: cliente, in/out e: estr) $\rightarrow$ res: itConj(promesa)

```
1  if ¬(e.ultimoLlamado.cliente = c ∧ e.ultimoLlamado.fueUltimo) then                O(1)
2    itClaves(diccString(infoTitulo)) it ← CrearIt(e.titulos)                        O(1)
3    conj(promesa) proms ← vacio()                                                   O(1)
4    tuplaPorClientes tup                                                            O(1)
5    while (HayMas?(it))                                              T*             O(1)
6      tup ← BuscarCliente(Significado(Nombre(Actual(it)), e.titulos).arrayClientes)
7                                                                     O(C*|nombre(actual(it))|)
8      if tup.promVenta ≠ NULL then AgregarRapido(proms, *(tup.promVenta))          O(1)
```

```
9       if tup.promCompra ≠ NULL then AgregarRapido(proms, *(tup.promCompra)) O(1)
10      Avanzar(it)                                                          O(1)
11    endWhile
12    e.ultimoLlamado.promesas ← proms                                       O(1)
13  fi
14  return(crearIt(e.ultimoLlamado.promesas))                                O(1)
```

Complejidad: 4*O(1)+T*(O(1)+O(C*|nombre(actual(it))|)+3*O(1))+O(1)+O(1)⊆O(T*C*|max_nt|)

iAccionesPorCliente(in c: cliente, in nt, nombreT, in e: estr) → res: nat

```
1   return(BuscarCliente(c, Significado(nt, e.titulos)).cantAcc)
```

Complejidad: O(log(C)+|nt|)

iInaugurarWolfie(in c: conj(cliente)) → res: estr

```
1   res.titulos ← CrearDicc()                                              O(1)
2   res.clientes ← NuevoConjEstNat(c)                                      O(C(log(C)))
3   res.ultimoLlamado ← <0, Vacio(), false>                                O(1)
```

Complejidad: O(C(log(C)))

iAgregarTítulo(in t: título, in/out e: estr) → res: nat

```
1   Definir(e.titulos, nombre(t), <CrearArrayClientes(CrearIt(e.clientes),cardinal
2         (e.clientes)), t, #maxAcciones(t))
```

Complejidad: O(|nombre(t)|+C)

iActualizarCotización(in nt: nombre, in cot: nat, in/out e: estr)

```
1   infoTitulo s ← Significado(nt, e.titulos)        O(|nt|)
2   recotizar(cot, s.titulo)
3   nat i ← 0                                                              O(1)
4   while i<|s.arrayClientes|                                              C*
5     if (s.arrayClientes[i].promVenta ≠ NULL \yluego limite(*(s.arrayClientes[i].promVenta))
6> cotizacion(s.titulo)) then
7       s.arrayClientes[i].cantAcc -= cantidad(*(s.arrayClientes[i].promVenta))   O(1)
8       s.accDisponibles += cantidad(*(s.arrayClientes[i].promVenta))             O(1)
9       s.arrayClientes[i].promVenta = NULL                                       O(1)
10    fi
11  endWhile
12  arreglo_dimensionable(tuplaPorCantAcc)[|s.arrayClientes|] arr                 O(C)
13  CambiarPorCantAcc(s.arrayClientes, arr)                                       O(C)
14  heapsort(arr)                                                                 O(C(log(C)))
15  i ← 0                                                                         O(1)
16  while i<|s.arrayClientes|                                                     C*
17    if (arr[i].promCompra ≠ NULL \yluego limite(*(arr[i].promCompra)) < cotizacion(s.titulo)
18 cantidad(*(arr[i].promCompra)) ≤ s.accDisponibles) then                       O(1)
19      arr[i].cantAcc += cantidad(*(arr[i].promCompra))                        O(1)
20      s.accDisponibles -= cantidad(*(arr[i].promCompra))                      O(1)
21      arr[i].promCompra = NULL                                                O(1)
22    fi
23    i++                                                                       O(1)
24  endWhile
25  CambiarPorCliente(arr, s.arrayClientes)                                     O(C)
26  heapsort(s.arrayClientes)                                                   O(C(log(C)))
```

Complejidad: O(|nt|)+2*O(1)+C*4*O(1)+O(C)+O(C)+O(C(log(C)))+O(1)+C*4*O(1)+O(C)+O(C(log(C)))=
O(|nt|+C(log(C)))

Aclaraciones: En este algoritmo hicimos un cambio de tipo de tupla a tuplaPorCantAcc, que es igual pero con otra relación de orden, por lo que al ordenar el arreglo, queda ordenado de mayor a menor por la cantidad de acciones de cada cliente. Luego de efectuar el cumplimiento de las promesas de compra, volvemos a cambiar a tuplaPorClientes y reordenamos en base a los clientes para poder seguir haciendo búsqueda logarítmica por clientes.

iAgregarPromesa(in c: cliente, in p:promesa, in/out e:estr)

```
1  promesa prom ← p                                                                    O(1)
2  if tipo(prom)=compra then                                                           O(1)
3   BuscarCliente(c, Significado(titulo(prom), e.titulos).arrayClientes).promCompra ← &prom
4                                                                                      O(|titulo(p)|+C)
5  else
6   BuscarCliente(c, Significado(titulo(prom), e.titulos).arrayClientes).promCompra ← &prom
7                                                                                      O(|titulo(p)|+C)
8  fi
```

Complejidad: O(1)+O(1)+O(|titulo(p)|+C)=O(|titulo(p)|+C)

iEnAlza(in nt: nombreT, in e: estr) → res: bool

```
1  return(enAlza(Significado(nt, e.titulos).titulo))
```

Complejidad: O(|nt|)

2.3.2. Algoritmos de itTítulos

iCrearIt(in e: estr) → res: iterador

```
1  return(<crearIt(e.titulos), &(e.titulos)>)
```

Complejidad: O(|nt|)

iActual(in i: iterador) → res: titulo

```
1  return(Significado(Actual(i.it), *(i.dicc)).titulo)
```

Complejidad: O(|nt|)

iPróximo(in/out i: iterador)

```
1  avanzar(i.it)
```

Complejidad: O(1)

iHayPróximo(in i: iterador) → res: bool

```
1  return(HayMas(i.it))
```

Complejidad: O(1)

2.3.3. Funciones auxiliares

CrearArrayClientes(in it: itConjEstNat, in n: nat) → res: arreglo_dimensionable(tuplaPorClientes)

```
1  arreglo_dimensionable(tuplaPorClientes)[n] arr    O(n)
2  nat i ← 0                                                          O(1)
3  do                                                                 n*
4    arr[i] ← <Actual(it), 0, NULL, NULL>                             O(1)
5    i++                                                              O(1)
6    Proximo(it)                                                      O(1)
7  while hayProx(it)                                                  O(1)
8  return arr
```

Complejidad: O(n)+O(1)+n*4*O(1)=O(n)

CambiarPorCantAcc(in a1: arreglo_dimensionable(tuplaPorCliente), in/out a2: arreglo_dimensionable(tuplaPorCantAcc))

```
1  nat i ← 0                                                          O(1)
2  while i<|a1|                                                       |a1|*
3    a2[i].cliente ← a1[i].cliente                                    O(1)
4    a2[i].cantAcc ← a1[i].cantAcc                                    O(1)
5    a2[i].promCompra ← a1[i].promCompra                              O(1)
6    a2[i].promVenta ← a1[i].promVenta                                O(1)
7    i++                                                              O(1)
8  endWhile
```

Complejidad: O(1)+|a1|*5*O(1)=O(|a1|)

CambiarPorCliente(in a1: arreglo_dimensionable(tuplaPorCantAcc), in/out a2: arreglo_dimensionable(tuplaPorCliente))

```
1  nat i ← 0                                                    O(1)
2  while i<|a1|                                                 |a1|*
3    a2[i].cliente ← a1[i].cliente                              O(1)
4    a2[i].cantAcc ← a1[i].cantAcc                              O(1)
5    a2[i].promCompra ← a1[i].promCompra                        O(1)
6    a2[i].promVenta ← a1[i].promVenta                          O(1)
7    i++                                                        O(1)
8  endWhile
```

Complejidad: O(1)+|a1|*5*O(1)=O(|a1|)

BuscarCliente(**in** cliente: cliente, **in** a: arreglo_dimensionable(tuplaPorCliente)) → **res** = tuplaPorCliente

```
1  int: arriba ← longitud(a)
2  int: abajo ← 0
3  int: centro
4  while (abajo ≤ arriba)
5          centro ← (arriba + abajo)/2;
6     if (arreglo[centro].Π1 == cliente)
7                  return a[centro];
8     else
9                  if (cliente < arreglo[centro].Π1)
10                         arriba ← centro−1;
11                 else
12                         abajo ← centro+1;
13                 endIf
14         endIf
15 endWhile
```

Complejidad O(log(|a|)) porque es una implementacion del algoritmo de búsqueda, que por lo visto en clase, tiene complejidad logarítmica en la longitud del arreglo.

2.4. Servicios Usados

Módulo	Operación	Complejidad Requerida
diccString(infoTitulo)	CrearIt	O(1)
diccString(infoTitulo)	Definir	|nt|
diccString(infoTitulo)	Significado	|nt|
conj(promesa)	Vacio	O(1)
conj(promesa)	AgregarRapido	O(1)
itClaves(diccString)	HayMás	O(1)
itClaves(diccString)	Actual	O(1)
itClaves(diccString)	Avanzar	O(1)
	BuscarCliente	O(log(C))
conjEstNat	NuevoConjEstNat	O(C(log(C)))
itConjEstNat	CrearIt	O(1)
itConjEstNat	HayProx	O(1)
itConjEstNat	Proximo	O(1)
itConjEstNat	Actual	O(1)
arreglo_dimensionable	CrearNuevo	O(n)
arreglo_dimensionable	AgregarElemento	O(1)
arreglo_dimensionable	•[•]	O(1)
	heapsort	O(n(log(n)))

3. Módulo Diccionario String(alpha)

El módulo Diccionario String provee un diccionario en el que la característica distintiva es que las claves definidas son strings, permitiendo esto que las operaciones de definición de claves en complejidad O(|clave|), como así también la obtención del valor asociado a la clave y el testeo de si dicha clave está definida. Para el recorrido sus elementos se provee de un iterador de Claves que permite recorrer las claves del Diccionario String.

3.1. Interfaz

3.1.1. Parámetros formales

género α
se explica con: DICCIONARIO(STRING, α), ITERADOR UNIDIRECCIONAL.

géneros: `diccString(`α`), itClaves(diccString)`.

3.1.2. Operaciones básicas de Diccionario String(α)

CREARDICC() $\rightarrow$ *res* : `diccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} vacio\}$
Complejidad: O(1)
Descripcion: Crea un diccionario vacío.

DEFINIR(**in/out** d: `diccString(`α`)`, **in** c: `string`, **in** s: α)
Pre $\equiv$ $\{d =_{\text{obs}} d_0 \wedge \neg def?(d, c)\}$
Post $\equiv$ $\{d =_{\text{obs}} definir(d_0, c, s)\}$
Complejidad: O($longitud(c)$)
Descripcion: Define la clave c con el significado s en el diccionario d.

DEFINIDO?(**in** d: `diccString(`α`)`, **in** c: `string`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} def?(c, d)\}$
Complejidad: O($longitud(c)$)
Descripcion: Devuelve true si y solo si c está definido como clave en el diccionario.

SIGNIFICADO(**in** d: `diccString(`α`)`, **in** c: `string`) $\rightarrow$ *res* : α
Pre $\equiv$ $\{def?(c, d)\}$
Post $\equiv$ $\{res =_{\text{obs}} obtener(c, d)\}$
Complejidad: O($longitud(c)$)
Descripcion: Devuelve el significado con clave c.
Aliasing: No se devuelve una copia del α en res, se devuelve una referencia a la original.

3.1.3. Operaciones básicas del iterador de claves de Diccionario String(α)

CREARIT(**in** d: `diccString(`α`)`) $\rightarrow$ *res* : `itClaves(`*string*`)`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ crearIt(d.claves) }
Complejidad: O(1)
Descripcion: Crea y devuelve un iterador de claves de Diccionario String.

HAYMAS?(**in** d: `itClaves(`*string*`)`) $\rightarrow$ *res* : *bool*
Pre $\equiv$ {true}

Post $\equiv$ {res $=_{\text{obs}}$ hayMas?(it)}
Complejidad: O($longitud(secuSuby(d))$)
Descripcion: Informa si hay más elementos por iterar.

ACTUAL(**in** d: `itClaves(`*string*`)`) $\rightarrow$ *res* : *string*
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ actual(it)}
Complejidad: O($longitud(secuSuby(d))$)
Descripcion: Devuelve la clave de la posición actual.

AVANZAR(**in/out** it: `itClaves(`*string*`)`)
Pre $\equiv$ {hayMas?(it) $\wedge$ $it = it_0$}
Post $\equiv$ {it $=_{\text{obs}}$ avanzar(it_0)}
Complejidad: O($longitud(secuSuby(d))$)
Descripcion: Avanza a la próxima clave.

3.2. Representación

3.2.1. Representación del Diccionario String(α)

`diccString`(α) **se representa con** `estrDic`

donde `estrDic` es `tupla`(*raiz*: `puntero(nodo)` *claves*: `lista(`*string*`)`)

`Nodo` **se representa con** `estrNodo`

donde `estrNodo` es `tupla`(*valor*: `puntero(`α`)` *hijos*: `arreglo_estático[256](puntero(nodo))`)

3.2.2. Invariante de Representación de diccString

(I) Existe un único camino entre cada nodo y el nodo raíz (no hay ciclos).

(II) Todos los nodos hojas, es decir, todos los que tienen su arreglo hijos con todas sus posiciones en NULL, tienen que tener un valor distinto de NULL.

(III) Raíz es distinto de NULL

(IV) En claves está el camino que se recorre desde la raź hasta cada nodo hoja.

Rep : estrDic $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
raíz != NULL $\wedge_{\text{L}}$ noHayCiclos(e) $\wedge$ todasLasHojasTienenValor(e) $\wedge$
hayHojas(e) $\Rightarrow$ |e.claves|>0 $\wedge$
($\forall$ c : string)(c $\in$ caminosANodos(e)) $\Rightarrow$ $\big((\exists$ i: nat)(i $\in$ {0..|e.claves|}) $\Rightarrow$ (e.claves[i] = c)$\big)$

3.2.3. Operaciones auxiliares del invariante de Representación

noHayCiclos : puntero(nodo) $\longrightarrow$ bool

noHayCiclos(n, p) $\equiv$ ($\exists$ n:nat)(($\forall$ c: string)(|s| = n $\Rightarrow$ leer(p, s) = NULL))

leer : puntero(nodo) × string $\longrightarrow$ bool

leer(p, s) $\equiv$ **if** vacia?(s) **then**
p → valor
else
if p → hijos[prim(s)] = NULL **then** NULL **else** leer(p → hijos[prim(s)], fin(s)) **fi**
fi

todosNull : arreglo(puntero(nodo)) $\longrightarrow$ bool

todosNull(a) $\equiv$ auxTodosNull($a, 0$)

auxTodosNull : arreglo(puntero(nodo)) × nat $\longrightarrow$ bool

auxTodosNull(a, i) $\equiv$ **if** i <|a| **then** a[i] == NULL $\wedge$ auxTodosNull($a, i+1$) **else** a[i].valor == NULL **fi**

esHoja : puntero(nodo) $\longrightarrow$ bool

esHoja(p) $\equiv$ **if** p == NULL **then** false **else** todosNull(p.hijos) **fi**

todasLasHojas : puntero(nodo) × nat $\longrightarrow$ conj(nodo)

todasLasHojas(p, n) $\equiv$ **if** p == NULL **then**
false
else
if esHoja(p) **then** Ag(*p, vacio) **else** auxTodasLasHojas((*p).hijos, 256) **fi**
fi

auxTodasLasHojas : arreglo(puntero(nodo)) × nat $\longrightarrow$ conj(nodo)

auxTodasLasHojas(a, n) $\equiv$ hojasDeHijos($a, n, 0$)

hojasDeHijos : arreglo(puntero(nodo)) × nat × nat $\longrightarrow$ conj(nodo)

hojasDeHijos(a, n, i) $\equiv$ **if** i = n **then** $\emptyset$ **else** todasLasHojas(a[i]) $\cup$ hojasDeHijos($a, n, (i+1)$) **fi**

todasLasHojasTienenValor : puntero(nodo) $\longrightarrow$ bool

todasLasHojasTienenValor(p) $\equiv$ auxTodasLasHojasTienenValor(todasLasHojas($p, 256$))

auxTodasLasHojasTienenValor : arreglo(puntero(nodo)) $\longrightarrow$ bool

auxTodasLasHojasTienenValor(a) $\equiv$ **if** |a| = 0 **then**
true
else
dameUno(a).valor != NULL $\wedge$ auxTodasLasHojasTienenValor(sinUno(a))
fi

hayHojas : puntero(nodo) $\longrightarrow$ bool

Dada una estructura, indica si en la misma existe algún nodo cuyo arreglo hijos tenga todas las posiciones NULL.

caminosANodos : puntero(nodo) $\longrightarrow$ conj(string)

Dada una estructura, devuelve un conjunto con el único camino existente entre la raiz y cada hoja. El camino se obtiene de agregar la posición del arreglo hijos por la cual hay que bajar en cada nivel de la estructura hasta llegar a la hoja, conviritendo en cada paso esa posición en un CHAR y juntándolos en un STRING.

3.2.4. Función de abstracción de diccString

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

Abs(e) $=_{obs}$ d: dicc(string,α) | ($\forall$ c:string)(definido?(c, d)) = ($\exists$ n: nodo)(n $\in$ todasLasHojas(e)) n.valor != NULL
$\wedge$ ($\exists$ i:nat)(i $\in$ {0..|e.claves|}) $\Rightarrow$ e.claves[i] = c $\wedge_{\text{L}}$ significado(c, d) = leer($e.clave$).valor

3.2.5. Representación del iterador de Claves del Diccionario String(α)

`itClaves(`*string*`)` **se representa con** `puntero(nodo)`

Su Rep y Abs son los de itSecu(α) definido en el apunte de iteradores.

3.3. Algoritmos

3.3.1. Algoritmos de Diccionario String

iCrearDicc() → **res** = estrDicc(α)

```
1  n ← nodo                                                    O(1)
2  n ← crearNodo()                                             O(1)
3  raiz ← *n                                                   O(1)
```

Complejidad: 3*O(1) = O(1)

iCrearNodo() → **res** = nodo

```
1  d : arreglo_estatico[256]                                   O(1)
2  i ← 0                                                       O(1)
3  while (i < 256)                                             256*
4          d[i] ← NULL                                         O(1)
5  endWhile
6  hijos ← d                                                   O(1)
7  valor ← NULL                                                O(1)
```

Complejidad:2*O(1) + 256*O(1) + 2*O(1) = O(1)

iDefinir(**in/out** estrDicc(α): d, **in** string: c, **in** alfa: s)

```
1  i ← 0                                                       O(1)
2  p ← d.raiz                                                  O(1)
3  while (i < (longitud(s)))                                   |s|*
4          if (p.hijos[ord(s[i])] == NULL)                     O(1)
5                  n: nodo ← crearNodo()                       O(1)
6                  p.hijos[ord(s[i])] ← *n                     O(1)
7          endIf
8  p ← p.hijos[ord(s[i])]                                      O(1)
9  i++                                                         O(1)
10 endWhile
11 p.valor ← a                                                 O(1)
12 agregarAdelante(hijos, c)                                   O(|s|)
```

Complejidad: 2*O(1) + |s|*5*O(1) + O(1) + O(|s|) = O(|s|)

iSignificado(**in** estrDicc(α): d, **in** string: c) → **res** = α

```
1  i ← 0                                                       O(1)
2  p ← d.raiz                                                  O(1)
3  while (i < (longitud(s)))                                   |s|*
4          p ← p.hijos[ord(s[i])]                              O(1)
5  i++                                                         O(1)
6  endWhile
7  return p.valor                                              O(1)
```

Complejidad: 2*O(1) + |s|*2*O(1) + O(1) = O(|s|)

iDefinido?(**in** estrDicc(α): d, **in** string: c) → **res** = bool

```
1  i ← 0                                                       O(1)
2  p ← d.raiz                                                  O(1)
3  while (i < (longitud(s)))                                   |s|*
4          if (p.hijos[ord(s[i])] != NULL)                     O(1)
5                  p ← p.hijos[ord(s[i])]                      O(1)
6                  i++ O(1)
7          else
8                  return false                                O(1)
9          endIf
10 endWhile
11 return p.valor != NULL                                      O(1)
```

Complejidad: 2*O(1) + |s|*3*O(1) + O(1) = O(|s|)

iClaves(**in** estrDicc(α): d) → **res** = lista_enlazada(string)

```
1 return itClaves(d)                                                    O(1)
```

Complejidad: O(1)

3.3.2. Algoritmos del iterador de claves del Diccionario String

Utiliza los mismos algoritmos que itSecu(α) definido en el apunte de iteradores.

3.4. Servicios Usados

Módulo	Operación	Complejidad Requerida
arreglo_estático	AgregarElemento	O(1)
arreglo_estático	•[•]	O(1)
lista	AgregarAdelante	O(copy(α))
lista	•[•]	O(1)

4. Módulo Conjunto Estático de Nats

El módulo conjunto estático de nats representa a un conjunto de naturales, con la diferencia de que no se le pueden agregar y quitar elementos, sino que se inicializa con una cantidad fija de naturales. Además, el iterador recorre el conjunto de menor a mayor.

4.1. Interfaz

géneros conjEstNat, itConjEstNat

Se explica con: Conjunto(nat), Iterador Unidireccional(nat).

4.1.1. Operaciones básicas de conjEstNat

NuevoConjEstNat(**in** c: `conj(nat)`) → res : `conjEstNat`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ c}
Complejidad: $O(n * (log(n)))$
Descripcion: Crea un conjunto estático de nats

Pertenece?(**in** n: `nat`, **in** c: `conjEstNat`) → res : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ n ∈ c}
Complejidad: $O(n)$
Descripcion: Pregunta si el elemento pertenece al conjunto

Cardinal(**in** c: `conjEstNat`) → res : `nat`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}} \#c$}
Complejidad: $O(n)$
Descripcion: Devuelve la cantidad de elementos que hay en el conjunto

4.1.2. Operaciones básicas de itConjEstNat

CrearIt(**in** c: `conjEstNat`) $\rightarrow$ res : `itConjEstNat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearItUni(c)}
Complejidad: $O(1)$
Descripcion: Devuelve un iterador unidireccional a un conjunto estático de nats

Actual(**in** i: `itConjEstNat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ actual(i)}
Complejidad: $O(1)$
Descripcion: Devuelve la posicion actual

Próximo(**in** i: `itConjEstNat`) $\rightarrow$ res : `itConjEstNat`
Pre $\equiv$ {hayMas?(i)}
Post $\equiv$ {$res =_{\text{obs}}$ avanzar(i)}
Complejidad: $O(1)$
Descripcion: Avanza el iterador

HayPróx?(**in** i: `itConjEstNat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$i_0 = i$}
Post $\equiv$ {$res =_{\text{obs}}$ hayMas?(i}
Complejidad: $O(1)$
Descripcion: Pregunta si hay mas elementos para iterar

4.2. Representación

4.2.1. Representación de conjEstNat

`conjEstNat` **se representa con** `array: arreglo_dimensionable(nat)`

4.2.2. Invariante de representación de conjEstNat

Rep: los elementos estan ordenados y no hay repeticiones

Rep : array $\longrightarrow$ bool

Rep(a) $\equiv$ true $\Longleftrightarrow$ ($\forall$ i: nat) (i <longitud(a)-1 $\Rightarrow$ (definido?(a, i) $\wedge$ definido?(a, $i+1$) $\wedge_{\text{L}} a[i]<a[i+1]$))

4.2.3. Función de abstracción de conjEstNat

Abs : array a $\longrightarrow$ conjEstNat {Rep(a)}

Abs(a) $=_{\text{obs}}$ c: conjEstNat | ($\forall$ n: nat) $n \in c \Leftrightarrow$ estáEnArray?(n, a, 0)

estáEnArray? : nat $\times$ arreglo_dimensionable(nat) $\times$ nat $\longrightarrow$ bool

estáEnArray(n, a, i) $\equiv$ **if** i= longitud(a)-1 **then** false **else** $a[i] = n \vee$ estáEnArray?($n, a, i+1$) **fi**

4.2.4. Representación de itConjEstNat

`itConjEstNat` **se representa con** `iterador`

donde `iterador` es `tupla`(*pos*: `nat`, *lista*: `puntero(arreglo_dimensionable(nat))`)

4.2.5. Invariante de representación de itConjEstNat

Rep : iterador $\longrightarrow$ bool

Rep(i) $\equiv$ true $\Longleftrightarrow$ $i.pos$<longitud(*($i.lista$))

4.2.6. Función de abstracción de itConjEstNat

Abs : iterador it $\longrightarrow$ itConjEstNat {Rep(it)}

Abs(it) $=_{obs}$ $iConj$: itConjEstNat | actual($iConj$) = $a[i]$ $\wedge$ hayPróx($iConj$)=($i.pos$<longitud(*($i.lista$))-1) $\wedge$
(hayPróx($i.Conj$)$\Rightarrow$próximo($iConj$)=Abs(<$i.pos + 1, i.lista$>))

4.3. Algoritmos

4.3.1. Algoritmos de conjEstNat

iNuevoConjEstNat(in c: conj(nat)) $\rightarrow$ res: array

```
1  itConj(nat) it ← crearIt(c))                                  O(1)
2  arreglo_dimensionable(nat)[cardinal(c)] a                     O(n)
3  nat i ← 0                                                     O(1)
4  while (HaySiguiente?(it))                                     n*
5          a[i] ← Siguiente(it)                                  O(1)
6          i++                                                   O(1)
7          Avanzar(it)                                           O(1)
8  endWhile
9  heapsort(a)                                                   O(n(log(n)))
10 return(a)
```

Complejidad: O(1)+O(n)+O(1)+n*(O(1)+O(1)+O(1))+O(n(log(n))) = O(n(log(n)))
Aclaraciones: Utilizamos el algoritmo HEAPSORT provisto en el apunte ALGORITMOS BÁSICOS, con las complejidades allí descriptas.

iPertenece(in n: nat, in c: array) $\rightarrow$ res: bool

```
1  bool b ← false                                                O(1)
2  nat i ← 0                                                     O(1)
3  while (i<|c|)                                                 n*
4          b ← (b ∨ c[i]=n)                                      O(1)
5          i++                                                   O(1)
6  endWhile
7  return(b)
```

Complejidad: O(1)+O(1)+n*(O(1)+O(1)) = O(n)

4.3.2. Algoritmos de itConjEstNat

iCrearIt(in a: array) $\rightarrow$ res: iterador

```
1  return (<0, &a>)
```

Complejidad: O(1)

```
iActual(in it: iterador) → res: nat
1 return (*(it.lista))[it.pos]
```

Complejidad: O(1)

```
iActual(in/out it: iterador)
1 return <it.pos+1, it.lista>
```

Complejidad: O(1)

```
iHayPróximo?(in it: iterador) → res: bool
1 return (it.pos+1<longitud(it.lista))
```

Complejidad: O(1)

4.4. Servicios Usados

Módulo	Operación	Complejidad Requerida
arreglo_estático	CrearNuevo	O(n)
arreglo_estático	AgregarElemento	O(1)
arreglo_estático	•[•]	O(1)
	heapsort	O(n(log(n)))

4.5. TAD Conjunto Estático de Nats

TAD Conjunto Estático de Nats

igualdad observacional
$(\forall c, c' : \text{conjEstNat})\ (c =_{\text{obs}} c' \iff ((\forall a : nat)(a \in c =_{\text{obs}} a \in c')))$

géneros conjEstNat

exporta conjEstNat, generadores, observadores, #

usa Bool, Nat, Conjunto(nat)

observadores básicos

• ∈ •	: nat × conjEstNat	⟶	bool

generadores

crearConjEstNat:	conj(nat)	⟶	conj(EstNat)

otras operaciones

#	: conj(EstNat)	⟶	nat

axiomas $\forall c$: conj(nat), $\forall n$: nat

$n \in \text{crearConjEstNat}(c) \equiv (n \in c)$

$\#(\text{crearConjEstNat}(c)) \equiv \#(c)$

Fin TAD

5. Módulo Promesa

5.1. Interfaz

géneros promesa

se explica con: PROMESA.

5.1.1. Operaciones básicas de promesa

TÍTULO(**in** p: `promesa`) $\rightarrow$ res : `nombre`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ título(p)}
Complejidad: $O(|titulo(p)|)$
Descripcion: Devuelve el nombre del título de la promesa

TIPO(**in** p: `promesa`) $\rightarrow$ res : `tipoPromesa`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ tipo(p)}
Complejidad: $O(1)$
Descripcion: Devuelve el tipo de promesa de la promesa

LIMITE(**in** p: `promesa`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ límite(p)}
Complejidad: $O(1)$
Descripcion: Devuelve el límite de la promesa

CANTIDAD(**in** p: `promesa`) $\rightarrow$ res : `cantidad`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ cantidad(p)}
Complejidad: $O(1)$
Descripcion: Devuelve la cantidad de acciones de la promesa

CREARPROMESA(**in** t: `nombre`, **in** $tipo$: `tipoPromesa`, **in** n: `dinero`, **in** m: `nat`) $\rightarrow$ res : `estr`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearPromesa(t, $tipo$, n, m)}
Complejidad: ($|t|$)
Descripcion: Devuelve una nueva promesa

5.2. Representación

5.2.1. Representación de promesa

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

donde `estr` es `tupla`(*título*: `nombre` *tipo*: `tipoPromesa` *límite*: `dinero` *cantidad*: `nat`)

5.2.2. Invariante de Representación de Promesa

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ true

5.2.3. Función de abstracción de Promesa

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

Abs(e) $=_{obs}$ p: promesa | título(p) = e.título $\wedge$ tipo(p) = e.tipo $\wedge$ límite(p) = e.límite $\wedge$ cantidad(p) = e.cantidad

5.3. Algoritmos

5.3.1. Algoritmos de promesa

iTitulo(in p: estr) → res = nombre

```
1 res ← e.titulo                                   O(|p.titulo|)
```

Complejidad: O(|p.titulo|)

iTipo(in p: estr) → res = tipoPromesa

```
1 res ← e.tipo                                     O(1)
```

Complejidad: O(1)

iLimite(in p: estr) → res = dinero

```
1 res ← e.limite                                   O(1)
```

Complejidad: O(1)

iCantidad(in p: estr) → res = nat

```
1 res ← e.cantidad                                 O(1)
```

Complejidad: O(1)

iCrearPromesa(in t: nombreT, in tipo: TipoPromesa, in n: dinero, in c: nat) → res = estr

```
1 res.titulo ← t                                   O(|p.titulo|)
2 res.tipo ← tipo                                  O(1)
3 res.limite ← n                                   O(1)
4 res.cantidad ← m                                 O(1)
```

Complejidad: 3*O(1) + O(|p.titulo|) = O(|p.titulo|)

6. Módulo Título

6.1. Interfaz

géneros título

se explica con: TÍTULO.

6.1.1. Operaciones básicas de título

NOMBRE(**in** t: `título`) → res : `nombre`
Pre ≡ {true}
Post ≡ {res $=_{obs}$ nombre(t)}
Complejidad: $O(|nombre(t)|)$
Descripcion: Devuelve el nombre del título

#MÁXACCIONES(**in** t: `título`) → res : `nat`
Pre ≡ {true}

Post $\equiv \{res =_{\text{obs}} \#\text{máxAcciones(t)}\}$
Complejidad: $O(1)$
Descripcion: Devuelve el máximo de cantidad de acciones

COTIZACIÓN(**in** t: `título`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{cotización(t)}\}$
Complejidad: $O(1)$
Descripcion: Devuelve la cotización del título

ENALZA(**in** t: `título`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{enAlza(t)}\}$
Complejidad: $O(1)$
Descripcion: Indica si el título está o no en alza

CREARTÍTULO(**in** t: `nombre`, **in** c: `dinero`, **in** n: `nat`) $\rightarrow$ res : `título`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{crearTítulo}(s, c, n)\}$
Complejidad: $O(|nombre(t)|)$
Descripcion: Devuelve una nuevo título

RECOTIZAR(**in** d: `dinero`, **in** t: `título`) $\rightarrow$ res : `título`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{recotizar}(d, t)\}$
Complejidad: $O(1)$
Descripcion: Cambia la cotización del título

6.2. Representación

6.2.1. Representación de título

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

donde `estr` es `tupla`(*nombre*: `nombre`, *#máxAcciones*: `nat`, *cotización*: `dinero`, *enAlza*: `bool`)

6.2.2. Invariante de Representación de Título

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ true

6.2.3. Función de abstracción de Título

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

Abs(e) $=_{\text{obs}}$ t: título | nombre(t) = e.nombre $\wedge$ #máxAcciones(t) = e.#máxAcciones $\wedge$ cotización(t) = e.cotización $\wedge$ enAlza(t) = e.enAlza

6.3. Algoritmos

6.3.1. Algoritmos de título

iNombre(in estr: t) → res = nombre

```
1 res ← e.nombre                                   O(|nombre(t)|)
```

Complejidad:O(|nombre(t)|)

i#máxAcciones(in estr: t) → res = nat

```
1 res ← e.#maxAcciones                             O(1)
```

Complejidad:O(1)

iCotización(in estr:: t) → res = dinero

```
1 res ← e.cotizacion                               O(1)
```

Complejidad:O(1)

iEnAlza(in estr: t) → res = bool

```
1 res ← e.enAlza                                   O(1)
```

Complejidad:O(1)

iCrearTítulo(in nombre: n, in nat: max, in dinero: c) → res = estr

```
1 res.nombre ← n                                   O(|nombre(t)|)
2 res.#maxAcciones ← max                           O(1)
3 res.enAlza ← true                                O(1)
4 res.cotizacion ← c                               O(1)
```

Complejidad: O(|nombre(t)|) + 3*O(1) = O(|nombre(t)|)

iRecotizar(in dinero: c, in/out estr: t)

```
1 t.enAlza ← (c>t.cotizacion)                      O(1)
2 t.cotizacion ← c                                 O(1)
```

Complejidad: 2*O(1) = O(1)