Algoritmos y Estructura de Datos II

Primer cuatrimestre 2014

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

Trabajo Practico 2

Grupo 10

Integrante	LU	Correo electrónico
Lucía, Parral	162/13	luciaparral@gmail.com
Nicolás, Roulet		
Pablo Nicolás, Gomez		
Guido Joaquin, Tamborindeguy		

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

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

2.1.1. Parámetros formales

géneros wolfie

se explica con: WOLFIE.

2.1.2. Operaciones básicas de wolfie

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

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

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

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

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

AGREGARTÍTULO(**in** t: `título`, **in/out** w: `wolfie`)
Pre $\equiv$ {$w_0 =_{\text{obs}} w \wedge$ ($\forall$ $t2$:título) ($t2 \in$ títulos(w) $\Rightarrow$ nombre(t) $\neq$ nombre($t2$)}
Post $\equiv$ {$w =_{\text{obs}}$ agregarTítulo(t, w_0}
Complejidad: $\Theta(|nombre(t)| + C)$

ACTUALIZARCOTIZACIÓN(**in** nt: `nombreTítulo`, **in** cot: `nat`, **in/out** w: `wolfie`) $\rightarrow$ res : `wolfie`
Pre $\equiv$ {$w_0 =_{\text{obs}} w \wedge$ ($\exists$ t:título) ($t \in$ títulos(w) $\wedge$ nombre(t) $=$ nt)}

Post $\equiv \{w =_{\text{obs}} \text{actualizarCotización}(nt, cot, w_0)\}$
Complejidad: $\Theta(C \cdot |nt| + C \cdot log(C))$
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: $\Theta(|título(p)| + log(C))$
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: $\Theta(|nt|)$
Descripcion: Dado un título, informa si está o no en alza.

2.2. Representación

2.2.1. Representación de wolfie

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

donde `estr` es `tupla`(*títulos*: `diccTrie(nombre, <`*arrayClientes*`: array_dimensionable(tuplaPorCliente),`
cot`: nat,`
enAlza`: bool,`
maxAcc`:nat,`
accDisponibles`: nat>),`
clientes: `conjEstNat(cliente)`
últimoLlamado: `<`*cliente*`: cliente,` *promesas*`:conj(promesa),` *fueUltimo*`: bool>))`

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 $a<b \Leftrightarrow a.cantAcc < b.cantAcc$

(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 y 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*.

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, obtener(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, obtener(t, $e.titulos$).$arrayClientes$) $\wedge_{\text{L}}$buscarCliente(c, obtener(t, $e.titulos$).$arrayClientes$).$promCompra$=p) $\Rightarrow_{\text{L}}$título($*p$)=t $\wedge$ tipo($*p$)=compra $\wedge$ (límite($*p$)>obtener(t, $e.titulos$).cot $\vee$ cantidad($*p$)>obtener(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, obtener(t, $e.titulos$).$arrayClientes$) $\wedge_{\text{L}}$buscarCliente(c, obtener(t, $e.titulos$).$arrayClientes$).$promVenta$=p) $\Rightarrow_{\text{L}}$(título($*p$)=t $\wedge$ tipo($*p$)=venta $\wedge$ límite($*p$) <obtener(t, $e.titulos$).cot)$\big)$ $\wedge$

(IV)($\forall$ nt: nombreT) (def?(nt, $e.titulos$) $\Rightarrow_{\text{L}}$(obtener(nt, $e.titulos$).$accDisponibles$ = obtener(nt, $e.titulos$).$maxAcc$ - sumaAccClientes(obtener(nt, $e.titulos$).$arrayClientes$, 0) $\wedge$ obtener(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($e.últimoLlamado.cliente$, obtener(título(p), $e.titulos$).$arrayClientes$).promCompra = p
else
buscarCliente($e.últimoLlamado.cliente$, obtener(título(p), $e.titulos$).$arrayClientes$).promVenta = p
fi$\big)\big)$

(VII)($\forall$ t: título) 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]$\big)$

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**

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

Abs(e) $=_{\text{obs}}$ w: wolfie | clientes(w)=$e.clientes$ $\wedge$ títulos(w)=????????? $\wedge$
($\forall$ c: cliente) promesasDe(c, w)=damePromesas(crearIt($e.titulos$), e, c) $\wedge$
accionesPorCliente(c, t, w)=buscarCliente(obtener($t, e.titulos$).$arrayClientes$).$cantAcc$

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

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

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

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

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

```
1  if ¬(e.ultimoLlamado.cliente = c ∧ e.ultimoLlamado.fueUltimo) then
2          itClaves(diccString) it ← crearIt(e.titulos)
3          conj(promesa) proms ← vacio()
4          tuplaPorClientes tup
5          while (HayMas?(it))
6                  tup ← BuscarCliente(Obtener(Nombre(Actual(it)), e.titulos).arrayClientes)
7                  if tup.promVenta ≠ NULL then AgregarRapido(proms, *(tup.promVenta))
8                  if tup.promCompra ≠ NULL then AgregarRapido(proms, *(tup.promCompra))
9                  Avanzar(it)
10         endWhile
11         e.ultimoLlamado.promesas ← proms
12 fi
13 return(crearIt(e.ultimoLlamado.promesas)
```

iAccionesPorCliente(in c: cliente, in nt, nombreT, in e: estr) $\rightarrow$ res = nat

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

iInaugurarWolfie(in c: conj(cliente)) $\rightarrow$ res = estr

```
1  res.titulos ← CrearDicc()
2  res.clientes ← CrearDicc()
3  res.ultimoLlamado ← <0, Vacio(), false>
```

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

```
1  promesa prom ← p
2  if tipo(prom)=compra then
3          BuscarCliente(c, Obtener(titulo(prom), e.titulos).arrayClientes).promCompra ← &prom
4  else
5          BuscarCliente(c, Obtener(titulo(prom), e.titulos).arrayClientes).promCompra ← &prom
6  fi
```

iEnAlza(in nt: nombreT, in e: estr) $\rightarrow$ res = bool

```
1  return(Obtener(nt, e.titulos).enAlza)
```

iAgregarTitulo(in t: titulo, in/out e: estr) $\rightarrow$ res = nat

```
1  Definir(e.titulos, nombre(t), <CrearArrayClientes
```

3. Módulo DiccionarioTrie(alpha)

3.1. Interfaz

3.1.1. Parámetros formales

géneros string, α

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

géneros: `diccString(`α`)`, `itDicc(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 Diccionario String.

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

AVANZAR(**in/out** it: `itClaves(`*string*`)`) $\rightarrow$ *res* : `itClaves(`α`)`

Pre $\equiv$ {hayMas?(it) $\wedge$ $it = it_0$}
Post $\equiv$ {res $=_{obs}$ avanzar(it_0)}
Complejidad: O($longitud(c)$)
Descripcion: Avanza a la próxima clave.

3.2. Representacion

3.2.1. Representación del Diccionario String(α)

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

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

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

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

(I) Existe un único camino entre cada nodo y el nodo raiz (es decir, 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) Raiz es distinto de NULL

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

Rep : estrDic $\longrightarrow$ bool

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

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 {0..|e.claves|}) e.claves[i] = c $\wedge_{\text{L}}$ significado(c, d) = leer($e.clave$).valor

3.2.2. Operaciones auxiliares del invatriante 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) $\times$ string $\longrightarrow$ bool

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

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

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

auxTodosNull : arreglo(puntero(nodo)) $\times$ 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) $\times$ 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)) $\times$ nat $\longrightarrow$ conj(nodo)

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

hojasDeHijos : arreglo(puntero(nodo)) $\times$ nat $\times$ 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

3.2.3. 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() $\rightarrow$ **res** = diccString(α)

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

Complejidad

iCrearNodo() $\rightarrow$ **res** = nodo

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

Complejidad

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

```
1 i ← 0
```

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

Complejidad

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

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

Complejidad

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

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

Complejidad

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

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

Complejidad

3.3.2. Algoritmos del iterador de claves del Diccionario String

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

Heapsort(**in** arreglo(tuplas): a, **in** int: n) → **res** = arreglo(tuplas)

```
1  fin ← (n-1)
2  while (end > 0)
3          swap(a[fin], a[0])
4          fin ← (fin - 1)
5          siftDown(a,0,fin)
6  endWhile
```

Complejidad

HEAPIFICAR(**in** arreglo(tuplas): a, **in** int: n) → **res** = arreglo(tuplas)

```
1  comienzo ← (parteEntera((n-2)/2))
2  while (comienzo > 0)
3          siftDown(a,comienzo,n-1)
4          comienzo ← comienzo - 1
5  endWhile
```

Complejidad

SIFTDOWN(**in** arreglo(tuplas): a, **in** int: comienzo, **in** int: fin) → **res** = arreglo(tuplas)

```
1  int: raiz
2  int: hijo
3  int: pasaMano
4  raiz ← comienzo
5  while ((raiz * 2)+ 1 ≤ fin)
6          hijo ← (raiz*2)+1
7          pasaMano ← raiz
8          if(a[pasaMano] < a[hijo])
9                  pasaMano ← hijo
10         endIf
11         if((hijo + 1 ≤ fin) && (a[pasaMano] < a[hijo+1]))
12                 pasaMano ← hijo + 1
13         endIf
14         if(pasaMano != raiz)
15                 swap(a[raiz], a[pasaMano])
16                 raiz ← pasaMano
17         endIf
18 endWhile
```

Complejidad

BUSQUEDABINARIA(**in** arreglo(tuplas): a, nat: cliente, nat: tam)

```
1  int: arriba ← tam-1
2  int: abajo ← 0
3  int: centro
4  while (abajo ≤ arriba)
5          centro ← (arriba + abajo)/2;
6      if (arreglo[centro].Π1 == cliente)
7                  return 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

4. Módulo Conjunto Estático de Nats

4.1. Interfaz

géneros conjEstNat, itConjEstNat

Se explica con: CONJUNTO(NAT), ITERADOR UNIDIRECCIONAL(NAT). **Usa**:

4.1.1. Operaciones básicas de conjEstNat

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

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

Cardinal(**in** n: `conjEstNat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ n $\in$ c}
Complejidad: $\Theta(n)$
Descripcion: Pregunta si el elemento pertenece al 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: $\Theta(1)$
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: $\Theta(1)$
Descripcion: Devuelve la posicion actual

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

HayPróx?(**in** i: `itConjEstNat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$i_0 = i$}
Post $\equiv$ {$res =_{\text{obs}}$ hayMas?(i}
Complejidad: $\Theta(1)$
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)`

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, $i+1$) $\wedge_{\text{L}} a[i]<a[i+1]$))

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**

4.2.2. Representación de itConjEstNat

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

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

Rep : iterador $\longrightarrow$ bool

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

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

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

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

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

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

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

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

```
1  return (<|c|, &c>)
```

iActual(in it: iterador) $\rightarrow$ res = nat

```
1  return *(it.lista)[it.pos]
```

iActual(in/out it: iterador)

```
1 return <it.pos+1, it.lista>
```

iHayPróximo?(in it: iterador) → res = bool

```
1 return (it.pos+1<longitud(it.lista))
```

Servicios usados: se utlilzan solo tipos basicos, incluidos arreglos y punteros.

5. Módulo Promesa