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. Módulo Wolfie

1.1. Interfaz

1.1.1. Parámetros formales

géneros wolfie

se explica con: WOLFIE.

1.1.2. Operaciones básicas de wolfie

CLIENTES(**in** w: `wolfie`) $\rightarrow$ res : `itUni(cliente)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearIt(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 : `itPromesa(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`) $\rightarrow$ res : `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 $\equiv$ {$w =_{\text{obs}}$ 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`) $\rightarrow$ res : `wolfie`
Pre $\equiv$ {$w_0 =_{\text{obs}} w \wedge$ ($\exists t$: título) ($t \in$ títulos(w) $\wedge$ nombre(t) = título(p)) $\wedge$ $c \in$ clientes(w) $\wedge_{\text{L}}$($\forall p2$: promesa) ($p2 \in$ promesasDe(c, w) $\Rightarrow$ (título(p) $\neq$ título($p2$) $\vee$ tipo(p) $\neq$ tipo($p2$))) $\wedge$ (tipo(p) = vender $\Rightarrow$ accionesPorCliente(c, 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.

1.2. Representación

1.2.1. 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, $e.titulos$).$arrayClientes$) $\wedge$ 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**

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 : itTrie?????? $\times$ 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

2. Módulo DiccionarioTrie(alpha)

2.1. Interfaz

2.1.1. Parámetros formales

géneros string, α

se explica con: DICCIONARIO.

géneros: `diccString(`α`)`.

2.1.2. Operaciones básicas de Diccionario String(α)

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

DEFINIR(**in/out** d: `diccString(`α`)`, **in** c: `string`, **in** s: α)
Pre $\equiv$ $\{d =_{\text{obs}} d_0 \wedge \neg definido?(d, c)\}$
Post $\equiv$ $\{d =_{\text{obs}} definir(d_0, c, s)\}$
Complejidad: $\Theta(longitud(c))$
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: $\Theta(longitud(c))$
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: $\Theta(longitud(c))$
Descripcion: Devuelve el significado con clave c.
Aliasing: No se devuelve una copia del α en res, se devuelve una referencia a la original.

2.2. Representacion

2.2.1. Representación del Diccionario String(α)

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

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

`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

Rep : estrDic $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
raiz != NULL $\wedge_{\text{L}}$ noHayCiclos(e) $\wedge$ todasLasHojasTienenValor(e)

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

Abs(e) $=_{\text{obs}}$ d: diccString(string,α) | ($\forall$ c:string)(definido?(c, d)) = ($\exists$ n: nodo)(n $\in$ todasLasHojas(e)) n.valor != NULL $\wedge_{\text{L}}$ significado(c, d) = leer($e.clave$).valor

2.2.2. 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 $\rightarrow$ arr[prim(s)] = NULL **then** NULL **else** leer(p $\rightarrow$ arr[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

2.3. Algoritmos

Algorithm 1 iCrearDicc

```
n : nodo
n ← crearNodo()
raiz ← *n
```

Algorithm 2 iCrearNodo()

```
d : arreglo_estatico[256]
i ← 0
while i < 256 do
    d[i] ← NULL
end while
hijos ← d
valor ← NULL
```

Algorithm 3 iDefinir

```
i ← 0
p ← d.raiz
while i < (longitud(s)) do
    if p.hijos[ord(s[i])] == NULL then
        n : nodo ← crearNodo()
        p.hijos[ord(s[i])] ← *n
    end if
    p ← p.hijos[ord(s[i])]
    i++
end while
p.valor ← a
```

Algorithm 4 iSignificado

```
i ← 0
p ← d.raiz
while i < (longitud(s)) do
    p ← p.hijos[ord(s[i])]
    i++
end while
return p.valor
```

Algorithm 5 iDefinido?

```
i ← 0
p ← d.raiz
while i < (longitud(s)) do
    if p.hijos[ord(s[i])]! = NULL then
        p ← p.hijos[ord(s[i])]
        i++
    else
return false
    end if
end while
return p.valor! = NULL
```