Algoritmos y Estructuras de Datos II

Trabajo Práctico 2

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

Segundo Cuatrimestre de 2014

Grupo 16

Apellido y Nombre	LU	E-mail
Juan Ernesto Rinaudo	864/13	jangamesdev@hotmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com
Fernando Frassia	340/13	ferfrassia@gmail.com

Reservado para la cátedra

Instancia	Docente que corrigió	Calificación
Primera Entrega		
Recuperatorio		

Índice

1. Tad Extendidos

1.1. Secu(α)

otras operaciones
elemDeSecu : Secu(α) s × Nat n $\longrightarrow$ RUR $\{\text{n} < \text{long}(s)\}$

axiomas
elemDeSecu(s, n) $\equiv$ **if** $n = 0$ **then** prim(s) **else** elemDeSecu(fin(s), n-1) **fi**

1.2. Mapa

observadores básicos
restricciones : Mapa m $\longrightarrow$ secu(restriccion)
nroConexion : estacion e_1 × estacion e_2 × Mapa m $\longrightarrow$ nat$\{e_1, e_2 \subset \text{estaciones(m)} \wedge_{\text{L}} \text{conectadas?}(e_1, e_2, m)\}$

axiomas
restricciones(vacio) $\equiv$ $\langle\ \rangle$
restricciones(agregar(e, m)) $\equiv$ restricciones(m)
restricciones(conectar(e_1, e_2, r, m)) $\equiv$ restricciones(m) $\circ$ r
nroConexion(e_1, e_2,conectar(e_3, e_4, m)) $\equiv$ **if** $((e_1 = e_3 \wedge e_2 = e_4) \vee (e_1 = e_4 \wedge e_2 = e_3))$ **then**
long(restricciones(m)) - 1
else
nroConexion(e_1, e_2, m) - 1
fi
nroConexion(e_1, e_2,agregar(e, m)) $\equiv$ nroConexion(e_1, e_2, m)

2. Mapa

Interfaz

se explica con: RED, ITERADOR UNIDIRECCIONAL(α).

géneros: `red, itConj(Compu)`.

Operaciones básicas de Red

COMPUTADORAS(**in** r : `red`) $\rightarrow$ res : `itConj(Compu)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearIt(computadoras(r))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve las computadoras de red.

CONECTADAS?(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$\{c_1, c_2\} \subseteq$ computadoras(r)}
Post $\equiv$ {$res =_{\text{obs}}$ conectadas?(r, c_1, c_2)}
Complejidad: $\mathcal{O}(|c_1| + |c_2|)$
Descripción: Devuelve el valor de verdad indicado por la conexión o desconexión de dos computadoras.

INTERFAZUSADA(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `interfaz`
Pre $\equiv$ {$\{c_1, c_2\} \subseteq$ computadoras(r) $\wedge_{\text{L}}$ conectadas?(r, c_1, c_2)}
Post $\equiv$ {$res =_{\text{obs}}$ interfazUsada(r, c_1, c_2)}
Complejidad: $\mathcal{O}(|c_1| + |c_2|)$
Descripción: Devuelve la interfaz que c_1 usa para conectarse con c_2

INICIARRED() $\rightarrow$ res : `red`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ iniciarRed()}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea una red sin computadoras.

AGREGARCOMPUTADORA(**in/out** r : `red`, **in** c : `compu`)
Pre $\equiv$ {$r_0 =_{\text{obs}} r \wedge \neg(c \in$computadoras($r$))}
Post $\equiv$ {$r =_{\text{obs}}$ agregarComputadora(r_0, c)}
Complejidad: $\mathcal{O}(|c|)$
Descripción: Agrega una computadora a la red.

CONECTAR(**in/out** r : `red`, **in** c_1 : `compu`, **in** i_1 : `interfaz`, **in** c_2 : `compu`, **in** i_2 : `interfaz`)
Pre $\equiv$ {$r_0 =_{\text{obs}}$ r $\wedge$ $\{c_1, c_2\} \subseteq$ computadoras(r) $\wedge$ ip(c_1) $\neq$ ip(c_2) $\wedge_{\text{L}} \neg$ conectadas?(r, c_1, c_2) $\wedge \neg$ usaInterfaz?(r, c_1, i_1) $\wedge \neg$ usaInterfaz?(r, c_2, i_2)}
Post $\equiv$ {$r =_{\text{obs}}$ conectar(r, c_1, i_1, c_2, i_2)}
Complejidad: $\mathcal{O}(|c_1| + |c_2|)$
Descripción: Conecta dos computadoras y les añade la interfaz correspondiente.

2.1. Representaciorepresentacionn

Representación

`red` **se representa con** `e_red`

donde `e_red` es `tupla`(*vecinosEInterfaces*: `diccString`(*compu*: `string`, `diccString`(*compu*: `string`, *interfaz*: `nat`))
, *deOrigenADestino*: `diccString`(*compu*: `string`, `diccString`(*compu*: `string`, *secu(compu)*: `secu(string)`))
, *computadoras*: `conj(compu)`)

2.2. InvRep y Abs

1. El conjunto de claves de "uniones" es igual al conjunto de estaciones "estaciones".
2. "#sendas" es igual a la mitad de las horas de "uniones".
3. Todo valor que se obtiene de buscar el significado del significado de cada clave de "uniones", es igual el valor hallado tras buscar en "uniones" con el sinificado de la clave como clave y la clave como significado de esta nueva clave, y no hay otras hojas ademas de estas dos, con el mismo valor.
4. Todas las hojas de "uniones" son mayores o iguales a cero y menores a "#sendas".
5. La longitud de "sendas" es mayor o igual a "#sendas".

Rep : e_mapa $\longrightarrow$ bool
Rep(m) $\equiv$ true $\Longleftrightarrow$
m.estaciones = claves(m.uniones) $\wedge$ 1.
m.#sendas = #sendasPorDos(m.estaciones, m.uniones) / 2 $\wedge$ m.#sendas $\leq$ long(m.sendas) $\wedge_{\text{L}}$ 2. 5.
($\forall$ e1, e2: string)(e1 $\in$ claves(m.uniones) $\wedge_{\text{L}}$ e2 $\in$ claves(obtener(e1, m.uniones)) $\Rightarrow_{\text{L}}$
e2 $\in$ claves(m.uniones) $\wedge_{\text{L}}$ e1 $\in$ claves(obtener(e2, m.uniones)) $\wedge_{\text{L}}$
obtener(e2, obtener(e1, m.uniones)) = obtener(e1, obtener(e2, m.uniones)) $\wedge$ 3. 4.
obtener(e2, obtener(e1, m.uniones)) < m.#sendas) $\wedge$
($\forall$ e1, e2, e3, e4: string)((e1 $\in$ claves(m.uniones) $\wedge_{\text{L}}$ e2 $\in$ claves(obtener(e1, m.uniones)) $\wedge$
e3 $\in$ claves(m.uniones) $\wedge_{\text{L}}$ e4 $\in$ claves(obtener(e3, m.uniones))) $\Rightarrow_{\text{L}}$
(obtener(e2, obtener(e1, m.uniones)) = obtener(e4, obtener(e3, m.uniones)) $\Longleftrightarrow$
(e1 = e3 $\wedge$ e2 = e4) $\vee$ (e1 = e4 $\wedge$ e2 = e3)))) 3.

#sendasPorDos : conj(α) c $\times$ dicc($\alpha \times$ dicc($\alpha \times \beta$)) d $\longrightarrow$ nat {c $\subset$ claves(d)}

#sendasPorDos(c, d) $\equiv$ **if** $\emptyset$?(c) **then**
0
else
#claves(obtener(dameUno(c),d)) + #sendasPorDos(sinUno(c), d)
fi

Abs : e_mapa m $\longrightarrow$ mapa {Rep(m)}
Abs(m) $=_{\text{obs}}$ p: mapa |
m.estaciones = estaciones(p) $\wedge_{\text{L}}$
($\forall$ e1, e2: string)((e1 $\in$ estaciones(p) $\wedge$ e2 $\in$ estaciones(p)) $\Rightarrow_{\text{L}}$
(conectadas?(e1, e2, p) $\Longleftrightarrow$
e1 $\in$ claves(m.uniones) $\wedge$ e2 $\in$ claves(obtener(e2, m.uniones)))) $\wedge_{\text{L}}$
($\forall$ e1, e2: string)((e1 $\in$ estaciones(p) $\wedge$ e2 $\in$ estaciones(p)) $\wedge_{\text{L}}$
conectadas?(e1, e2, p) $\Rightarrow_{\text{L}}$
(restriccion(e1, e2, p) = m.sendas[obtener(e2, obtener(e1, m.uniones))] $\wedge$ nroConexion(e1, e2, m) = obtener(e2, obtener(e1, m.uniones))) $\wedge$ long(restricciones(p)) = m.#sendas $\wedge_{\text{L}}$ ($\forall$ n:nat) (n <m.#sendas $\Rightarrow_{\text{L}}$ m.sendas[n] = ElemDeSecu(restricciones(p), n)))

2.3. Algoritmos

Algoritmos

iComputadoras(**in** r : `red`) $\rightarrow$ res : `itConj(Compu)`
1: $res \leftarrow$ CrearIt($r.computadoras$) $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iConectadas?(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `bool`

1: $res \leftarrow$ Definido?(Significado($r.vecinosEInterfaces$, c_1), c_2) $\mathcal{O}(|c_1| + |c_2|)$

Complejidad: $\mathcal{O}(|c_1| + |c_2|)$

iInterfazUsada(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `interfaz`

1: $res \leftarrow$ Significado(Significado($r.vecinosEInterfaces$, c_1), c_2) $\mathcal{O}(|c_1| + |c_2|)$

Complejidad: $\mathcal{O}(|c_1| + |c_2|)$

iIniciarRed() $\rightarrow$ res : `red`

1: $res \leftarrow$ tupla($vecinosEInterfaces$: Vacío(), $deOrigenADestino$: Vacío(), $computadoras$: Vacío()) $\mathcal{O}(1+1+1)$

Complejidad: $\mathcal{O}(1)$

$\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) =$
$3 * \mathcal{O}(1) = \mathcal{O}(1)$

iAgregarComputadora(**in/out** r : `red`, **in** c : `compu`)

1: Agregar($r.computadoras$, c) $\mathcal{O}(1)$
2: Definir($r.vecinosEInterfaces$, c, Vacío()) $\mathcal{O}(|c|)$
3: Definir($r.deOrigenADestino$, c, Vacío()) $\mathcal{O}(|c|)$

Complejidad: $\mathcal{O}(|c|)$

$\mathcal{O}(1) + \mathcal{O}(|c|) + \mathcal{O}(|c|) =$
$2 * \mathcal{O}(|c|) = \mathcal{O}(|c|)$

iConectar(**in/out** r : `red`, **in** c_1 : `compu`, **in** i_1 : `interfaz`, **in** c_2 : `compu`, **in** i_2 : `interfaz`)

1: Definir(Significado($r.vecinosEInterfaces$, c_1), c_2, i_1) $\mathcal{O}(|c_1| + |c_2| + 1)$
2: Definir(Significado($r.vecinosEInterfaces$, c_2), c_1, i_2) $\mathcal{O}(|c_2| + |c_1| + 1)$
3:

Complejidad: $\mathcal{O}(|e_1| + |e_2|)$

$\mathcal{O}(|e_1| + |e_2|) + \mathcal{O}(|e_1| + |e_2|) + \mathcal{O}(1) + \mathcal{O}(1) =$
$2 * \mathcal{O}(1) + 2 * \mathcal{O}(|e_1| + |e_2|) =$
$2 * \mathcal{O}(|e_1| + |e_2|) = \mathcal{O}(|e_1| + |e_2|)$

3. Ciudad Robótica

Interfaz

se explica con: Ciudad Robótica, Iterador Unidireccional(α).

géneros: `ciudad, itRURs`.

Operaciones básicas de Ciudad Robótica

PróximoRUR(**in** c: `ciudad`) $\rightarrow$ res : `rur`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ próximoRUR(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el PróximoRUR de una ciudad, esto es, de añadirse un robot se le asignaría este RUR.

Mapa(**in** c: `ciudad`) $\rightarrow$ res : `mapa`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ mapa(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el mapa de la ciudad.

Robots(**in** c: `ciudad`) $\rightarrow$ res : `itRURs`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crearIt(robots(c))}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un iterador de los robots de la ciudad.

Estación(**in** u: `rur`, **in** c: `ciudad`) $\rightarrow$ res : `estacion`
Pre $\equiv$ {$u \in$ robots(c)}
Post $\equiv$ {$res =_{\text{obs}}$ estación(u, c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la estación en la cual está el robot.

Tags(**in** u: `rur`, **in** c: `ciudad`) $\rightarrow$ res : `conj(tags)`
Pre $\equiv$ {$u \in$ robots(c)}
Post $\equiv$ {$res =_{\text{obs}}$ tags(u, c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve los tags del robot.

#Infracciones(**in** u: `rur`, **in** c: `ciudad`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$u \in$ robots(c)}
Post $\equiv$ {$res =_{\text{obs}}$ #infracciones(u, c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de infracciones cometidas por el robot.

Crear(**in** m: `mapa`) $\rightarrow$ res : `ciudad`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ crear(m)}
Complejidad: $\mathcal{O}(Cardinal(Estaciones(m)) * |e_m|)$
Descripción: Crea una ciudad con un mapa y sin robots.

Entrar(**in** ts: `conj(tags)`, **in** e: `estación`, **in/out** c: `ciudad`)
Pre $\equiv$ {$c_0 \equiv c \wedge$ e $\in$ estaciones(c_0)}
Post $\equiv$ {$c =_{\text{obs}}$ entrar(ts, e, c_0)}
Complejidad: $\mathcal{O}(log_2 N + |e| + S * R)$
Descripción: Añade un robot a la ciudad, le asigna el próximoRUR y sus infracciones son nulas.

Mover(**in** u: `rur`, **in** e: `estación`, **in/out** c: `ciudad`)
Pre $\equiv$ {($c_0 \equiv c \wedge u \in$ robots(c_0) $\wedge e \in$ estaciones(c_0)) $\wedge_{\text{L}}$ conectadas?(estación(u, c_0), e mapa(c_0))}
Post $\equiv$ {$c =_{\text{obs}}$ mover(u, e, c_0)}

Complejidad: $\mathcal{O}(|e| + |e_0| + log_2 N_e + log_2 N_{e0})$
Descripción: Mueve un robot desde donde está a la estación indicada.

INSPECCIÓN(**in** e: `estación`, **in/out** c: `ciudad`)
Pre $\equiv \{c_0 \equiv c \wedge e \in \text{estaciones}(c_0)\}$
Post $\equiv \{c =_{\text{obs}} \text{inspeccion}(e, c_0)\}$
Complejidad: $\mathcal{O}(log_2 N)$
Descripción: Realiza la inspección de la estación indicada, remueve el robot con mayor cantidad de infracciones.

Operaciones del iterador

CREARIT(**in** c: `ciudad`) $\rightarrow$ res : `itRURs`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ CrearItUni(robots(c))}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea el iterador de robots.

ACTUAL(**in** it: `itRURs`) $\rightarrow$ res : `rur`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ Actual(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el actual del iterador de robots.

AVANZAR(**in** it: `itRURs`) $\rightarrow$ res : `itRURs`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ Avanzar(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Avanza el iterador de robots.

HAYMAS?(**in** it: `itRURs`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ HayMas?(it)}
Complejidad: $\mathcal{O}(1)$
Descripción: Se fija si hay mas elementos en el iterador de robots.

3.1. Representacion

Representación

`ciudad` **se representa con** `e_cr`

donde `e_cr` es `tupla`(*mapa*: `mapa`, *RUREnEst*: `diccString`(*estacion*: `string`, *robs*: `colaP`(*id*: `nat`, *inf*: `nat`)), *RURs*: `vector de tupla`(*id*: `nat`, *esta?*: `bool`, *e*: `string`, *inf*: `nat`, *carac*: `conj(string)`, *sendEv*: `arreglo_dimensionable de bool`), *#RURHistoricos*: `nat`)

3.2. InvRep y Abs

1. El conjunto de estaciones de 'mapa' es igual al conjunto con todas las claves de 'RURenEst'.
2. La longitud de 'RURs' es mayor o igual a '#RURHistoricos'.
3. Todos los elementos de 'RURs' cumplen que su primer componente ('id') corresponde con su posicion en 'RURs'. Su Componente 'e' es una de las estaciones de 'mapa', su componente 'esta?' es true si y solo si hay estaciones tales que su valor asignado en 'uniones' es igual a su indice en 'RURs'. Su Componente 'inf' puede ser mayor a cero solamente si hay algun elemento en 'sendEv' tal que sea false. Cada elemento de 'sendEv' es igual a verificar 'carac' con la estriccion obtenida al buscar el elemento con la misma posicion en la secuencia de restricciones de 'mapa'.

4. Cada valor contenido en la cola del significado de cada estacion de las claves de 'uniones' pertenecen unicamente a la cola asociada a dicha estacion y a ninguna otra de las colas asociadas a otras estaciones. Y cada uno de estos valores es menor a '#RURHistoricos' y mayor o igual a cero. Ademas la componente 'e' del elemento de la posicion igual a cada valor de las colas asociadas a cada estacion, es igual a la estacion asociada a la cola a la que pertenece el valor.

Rep : e_cr $\longrightarrow$ bool

Rep(c) $\equiv$ true $\Longleftrightarrow$ claves(c.RURenEst) = estaciones(c.mapa) $\wedge$ 1
#RURHistoricos $\leq$ Long(c.RURs) $\wedge_{\text{L}}$ ($\forall$ i:Nat, t:<id:Nat, esta?:Bool, e:String, 2
inf:Nat, carac:Conj(Tag), sendEv: ad(Bool)>)
(i<#RURHistoricos $\wedge_{\text{L}}$ ElemDeSecu(c.RURs, i) = t $\Rightarrow_{\text{L}}$ (t.e $\in$ estaciones(c.mapa) 3
$\wedge$ t.id = i $\wedge$ tam(t.sendEv) = long(Restricciones(c.mapa)) $\wedge$
(t.inf >0 $\Rightarrow$ ($\exists$ j:Nat) (j <tam(t.sendEv) $\wedge_{\text{L}}$ $\neg$ (t.sendEv[j]))) $\wedge$
(t.esta? $\Leftrightarrow$ ($\exists$ e1: String) (e1 $\in$ claves(c.RURenEst) $\wedge_{\text{L}}$ estaEnColaP?(obtener(e1, c.RURenEst), t.id)))
$\wedge$ ($\forall$ h : Nat) (h <tam(t.sendEv) $\Rightarrow_{\text{L}}$
t.sendEv[h] = verifica?(t.carac, ElemDeSecu(Restricciones(c.mapa), h))))) $\wedge_{\text{L}}$
($\forall$ e1, e2: String)(e1 $\in$ claves(c.RURenEst) $\wedge$ e2 $\in$ claves(c.RURenEst) $\wedge$ e1 $\neq$ e2 $\Rightarrow_{\text{L}}$ 4
($\forall$ n:Nat)(estaEnColaP?(obtener(e1, c.RURenEst), n) $\Rightarrow$ $\neg$ estaEnColaP?(obtener(e2, c.RURenEst), n)
$\wedge$ n <#RURHistoricos $\wedge_{\text{L}}$ ElemDeSecu(c.RURs, n).e = e1))

estaEnColaP? : ColaPri $\times$ Nat $\longrightarrow$ Bool

estaEnColaP?(cp, n) $\equiv$ **if** vacia?(cp) **then**
false
else
if desencolar(cp) = n **then**
true
else
estaEnColaP?(Eliminar(cp, desencolar(cp)), n)
fi
fi

Abs : e_cr c $\longrightarrow$ ciudad {Rep(c)}

Abs(c) $=_{\text{obs}}$ u: ciudad |
c.#RURHistoricos = ProximoRUR(U) $\wedge$ c.mapa = mapa(u) $\wedge_{\text{L}}$
robots(u) = RURQueEstan(c.RURs) $\wedge_{\text{L}}$
($\forall$ n:Nat) (n $\in$ robots(u) $\Rightarrow_{\text{L}}$ estacion(n,u) = c.RURs[n].e $\wedge$
tags(n,u) = c.RURs[n].carac $\wedge$ #infracciones(n,u) = c.RURs[n].inf)

RURQueEstan : secu(tupla) $\longrightarrow$ Conj(RUR)

tupla es <id:Nat, esta?:Bool, inf:Nat, carac:Conj(tag), sendEv:arreglo dimensionable(bool)>

RURQueEstan(s) $\equiv$ **if** vacia?(s) **then**
$\emptyset$
else
if Π_2(prim(fin(s))) **then**
Π_1(prim(fin(s))) $\cup$ RURQueEstan(fin(s))
else
RURQueEstan(fin(s))
fi
fi

`it` **se representa con** `e_it`

donde `e_it` es `tupla`(i: `nat`, $maxI$: `nat`, $ciudad$: `puntero(ciudad)`)

Rep : e_it $\longrightarrow$ bool
Rep(it) $\equiv$ true $\iff$ it.i $\leq$ it.maxI $\wedge$ maxI = ciudad.#RURHistoricos
Abs : e_it u $\longrightarrow$ itUni(α) {Rep(u)}
Abs(u) $=_{\text{obs}}$ it: itUni(α) | (HayMas?(u) $\wedge_{\text{L}}$ Actual(u) = ciudad.RURs[it.i] $\wedge$ Siguientes(u, $\emptyset$) = VSiguientes(ciudad, it.i++, $\emptyset$) $\vee$ ($\neg$HayMas?(u))

Siguientes : itUniu $\times$ conj(RURs)cr $\longrightarrow$ conj(RURs)

Siguientes(u, cr) $\equiv$ **if** HayMas(u)? **then** Ag(Actual(Avanzar(u)), Siguientes(Avanzar(u), cr)) **else** Ag($\emptyset$, cr) **fi**

VSiguientes : ciudadc $\times$ Nati $\times$ conj(RURs)cr $\longrightarrow$ conj(RURs)

VSiguientes(u, i, cr) $\equiv$ **if** i <c.#RURHistoricos **then** Ag(c.RURs[i], VSiguientes(u, i++, cr))) **else** Ag($\emptyset$, cr) **fi**

3.3. Algoritmos

Algoritmos

iPróximoRUR(**in** c: `ciudad`) $\rightarrow$ res : `rur`
1: $res \leftarrow (c.\#RURHistoricos)$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iMapa(**in** c: `ciudad`) $\rightarrow$ res : `mapa`
1: $res \leftarrow c.mapa$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iRobots(**in** c: `ciudad`) $\rightarrow$ res : `itRobots`
1: $res \leftarrow$ CrearIt($c.RURs$) $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iEstación(**in** u: `rur`, **in** c: `ciudad`) $\rightarrow$ res : `estación`
1: $res \leftarrow (c.RURs[u]).estacion$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iTags(**in** u: `rur`, **in** c: `ciudad`) $\rightarrow$ res : `conj(tags)`
1: $res \leftarrow (c.RURs[u]).carac$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iNFRACCIONES(**in** u: `rur`, **in** c: `ciudad`) $\rightarrow$ res : `nat`

1: $res \leftarrow (c.RURs[u]).inf$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iCREAR(**in** m: `mapa`) $\rightarrow$ res : `ciudad`

1: $res \leftarrow$ tupla($mapa$: m, $RUREnEst$: Vacío(), $RURs$: Vacía(), $\#RURHistoricos$: 0) $\mathcal{O}(1)$
2: var it:itConj(Estacion) $\leftarrow$ Estaciones(m) $\mathcal{O}(1)$
3: **while** HaySiguiente(it) **do** $\mathcal{O}(1)$
4: Definir($res.RUREnEst$, Siguiente(it), Vacío()) $\mathcal{O}(|e_m|)$
5: Avanzar(it) $\mathcal{O}(1)$
6: **end while**

Complejidad: $\mathcal{O}(Cardinal(Estaciones(m)) * |e_m|)$

$\mathcal{O}(1) + \mathcal{O}(1) + \sum_{i=1}^{Cardinal(Estaciones(m))}(\mathcal{O}(|e_m|) + \mathcal{O}(1)) =$
$2 * \mathcal{O}(1) + Cardinal(Estaciones(m)) * (\mathcal{O}(|e_m|) + \mathcal{O}(1)) =$
$Cardinal(Estaciones(m)) * (\mathcal{O}(|e_m|))$

iENTRAR(**in** ts: `conj(tags)`, **in** e: `string`, **in/out** c: `ciudad`)

1: Agregar(Significado($c.RUREnEst$, e), 0, $c.\#RURHistoricos$) $\mathcal{O}(log_2 n + |e|)$
2: Agregar($c.RURs$, $c.\#RURHistoricos$, tupla(id: $c.\#RURHistoricos$, $esta?$: $true$, $estacion$: e, inf: 0, $carac$: ts, $sendEv$: EvaluarSendas(ts, $c.mapa$)) $\mathcal{O}(1 + S * R)$
3: $c.\#RURHistoricos++$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(log_2 n + |e| + S * R)$

$\mathcal{O}(log_2 n + |e|) + \mathcal{O}(1 + S * R) + \mathcal{O}(1) = \mathcal{O}(log_2 n + |e| + S * R)$

iMOVER(**in** u: `rur`, **in** e: `estación`, **in/out** c: `ciudad`)

1: Eliminar(Significado($c.RUREnEst$, $c.RURs[u].estacion$), $c.RURs[u].inf$, u) $\mathcal{O}(|e| + log_2 N_{e0})$
2: Agregar(Significado($c.RUREnEst$, e), $c.RURs[u].inf$, u) $\mathcal{O}(|e| + log_2 N_e)$
3: **if** $\neg(c.RURs[u].sendEv$[NroConexion($c.RURs[u].estacion$, e, $c.mapa$)]) **then** $\mathcal{O}(|e_0| + |e|)$
4: $c.RURs[u].inf++$ $\mathcal{O}(1)$
5: **end if**
6: $c.RURs[u].estacion \leftarrow e$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(|e| + log_2 N_e)$

$\mathcal{O}(|e| + log_2 N_{e_0}) + \mathcal{O}(|e| + log_2 N_e) + \mathcal{O}(|e_0|, |e|) + max(\mathcal{O}(1), \mathcal{O}(0)) + \mathcal{O}(1) =$
$\mathcal{O}(2 * |e| + log_2 N_e + log_2 N_{e0}) + \mathcal{O}(|e_0| + |e|) + 2 * \mathcal{O}(1) =$
$\mathcal{O}(|e| + log_2 N_e + log_2 N_{e0}) + \mathcal{O}(|e_0| + |e|) =$
$\mathcal{O}(2 * |e| + |e_0| + log_2 N_e + log_2 N_{e0}) = \mathcal{O}(|e| + |e_0| + log_2 N_e + log_2 N_{e0})$ Donde e_0 es c.RURs[u]estacion antes de modificar el valor

iINSPECCIÓN(**in** e: `estación`, **in/out** c: `ciudad`)

1: var rur: nat $\leftarrow$ Desencolar(Significado($c.RUREnEst$, e)) $\mathcal{O}(log_2 N)$
2: $c.RURs[rur].esta? \leftarrow false$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(log_2 N)$

$\mathcal{O}(log_2 N) + \mathcal{O}(1) = \mathcal{O}(log_2 N)$

icrearIt(**in** c: `ciudad`) $\rightarrow$ res : `itRURs`

1: $itRURS \leftarrow \text{tupla}(i : 0, maxI : c.\#RURHistoricos, ciudad : \&c)$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iActual(**in** it: `itRURs`) $\rightarrow$ res : `rur`

1: $res \leftarrow (it.ciudad \rightarrow RURs)[it.i]$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iAvanzar(**in** it: `itRURs`) $\rightarrow$ res : `itRURs`

1: $it.i++$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iHayMas?(**in** it: `itRURs`) $\rightarrow$ res : `bool`

1: $res \leftarrow (it.i < it.maxI)$ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

4. Diccionario String(α)

Interfaz

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

se explica con: DICCIONARIO(STRING, α).

géneros: `diccString(`α`)`.

Operaciones básicas de Restricción

VACÍO() $\rightarrow$ res : `diccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacio()}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea nuevo diccionario vacio.

DEFINIR(**in/out** d: `diccString(`α`)`, **in** clv: `string`, **in** def: α)
Pre $\equiv$ {$d_0 =_{\text{obs}} d$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(clv, def, d)}
Complejidad: $\mathcal{O}(|clv|)$
Descripción: Agrega un nueva definicion.

DEFINIDO?(**in** d: `diccString(`α`)`, **in** clv: `string`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(clv, d)}
Complejidad: $\mathcal{O}(|clv|)$
Descripción: Revisa si la clave ingresada se encuentra definida en el Diccionario.

SIGNIFICADO(**in** d: `diccString(`α`)`, **in** clv: `string`) $\rightarrow$ res : `diccString(`α`)`
Pre $\equiv$ {def?(d, clv)}
Post $\equiv$ {$res =_{\text{obs}}$ obtener(clv, d)}
Complejidad: $\mathcal{O}(|clv|)$
Descripción: Devuelve la definicion correspondiente a la clave.

4.1. Representacion

Representación

Esta no es la version posta de la descripcion, es solo un boceto.

Para representar el diccionario de Trie vamos a utilizar una estructura que contiene el primer Nodo y la cantidad de Claves en el diccionario. Para los nodos se utilizo una estructura formada por una tupla, el primer elemento es el significado de la clave y el segundo es un arreglo de 256 elementos que contiene punteros a los hijos del nodo (por todos los posibles caracteres ASCII).

Para conseguir el numero de orden de un char tengo las funciones ord.

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

donde `e_nodo` es `tupla(`*definicion*: `puntero(`α`)`, *hijos*: `arreglo[256] de puntero(e_nodo))`

4.2. InvRep y Abs

1. Para cada nodo del arbol, cada uno de sus hijos que apunta a otro nodo no nulo, apunta a un nodo diferente de los apuntados por sus hermanos
2. A donde apunta el significado de cada nodo es distinto de a donde apunta el significado del resto de los nodos, con la excepcion que el significado apunta a "null"
3. No puden haber ciclos, es decir, que todos lo nodos son apuntados por un unico nodo del arbol, con la excepcion de la raiz, este no es apuntado por ninguno de los nodos del arbol
4. Debe existir aunque sea un nodo en el ultimo nivel, tal que su significado no apunta a "null"

Abs : e_nodo d $\longrightarrow$ diccString {Rep(d)}
Abs(d) $=_{obs}$ n: diccString |
(∀ n:e_nodo) Abs(n) $=_{obs}$ d: diccString | (∀ s:string) (def?(s, d) $\Rightarrow_L$ ((obtenerDelArbol(s, n) $\neq$ NULL $\wedge_L$ *(obtenerDelArbol(s, n) = obtener(s, d))))) $\wedge_L$

obtenerDelArbol : strings s × e_nodo$n$ $\longrightarrow$ puntero(α)

obtenerDelArbol(s, n) $\equiv$ **if** Vacia?(s) **then**
n.significado
else
if n.hijos[ord(prim(s)] = NULL **then**
NULL
else
obtenerDelArbol(fin(s), n.hijos[ord(prim(s))])
fi
fi

4.3. Algoritmos

Algoritmos

iVacío() → res : `diccString(`α`)`

1: res ← iNodoVacío() $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

iNodoVacío() → res : `e_nodo)`

1: res ← tupla($definicin$: NULL, $hijos$:arreglo[256] de puntero(e_nodo)) $\mathcal{O}(1)$
2: **for** var i:nat ← 0 to 255 **do** $\mathcal{O}(1)$
3: $res.hijos[i]$ ← NULL; $\mathcal{O}(1)$
4: **end for**

Complejidad: $\mathcal{O}(1)$

$\mathcal{O}(1) + \sum_{i=1}^{255} * \mathcal{O}(1) =$
$\mathcal{O}(1) + 255 * \mathcal{O}(1) =$
$256 * \mathcal{O}(1) = \mathcal{O}(1)$

iDefinir(**in/out** d: `diccString(`α`)`, **in** clv: `string`, **in** def : α)

1: var $actual$:puntero(e_nodo) ← &(d) $\mathcal{O}(1)$
2: **for** var i:nat ← 0 to LONGITUD(clv) **do** $\mathcal{O}(1)$
3: **if** $actual$ → $hijos$[ord($clv[i]$)] $=_{obs}$ NULL **then** $\mathcal{O}(1)$
4: $actual$ → ($hijos$[ord($clv[i]$)] ← &(iNodoVacío())) $\mathcal{O}(1)$

5: **end if**
6: $actual \leftarrow (actual \rightarrow hijos[\text{ord}(clv[i])])$ $\mathcal{O}(1)$
7: **end for**
8: $(actual \rightarrow definicion) \leftarrow \&(\text{Copiar}(def))$ $\mathcal{O}(1)$

Complejidad: |clv|

$\mathcal{O}(1) + \sum_{i=1}^{|clv|} max(\sum_{i=1}^{2} \mathcal{O}(1), \sum_{i=1}^{3} \mathcal{O}(1)) + \mathcal{O}(1) =$
$2 * \mathcal{O}(1) + |clv| * max(2 * \mathcal{O}(1), 3 * \mathcal{O}(1)) =$
$2 * \mathcal{O}(1) + |clv| * 3 * \mathcal{O}(1) =$
$2 * \mathcal{O}(1) + 3 * \mathcal{O}(|clv|) =$
$3 * \mathcal{O}(|clv|) = \mathcal{O}(|clv|)$

iDefinido?(**in** d: `diccString(`α`)`, **in** def: α) $\rightarrow$ res : `bool`

1: var $actual$:puntero(e_nodo) $\leftarrow$ &(d) $\mathcal{O}(1)$
2: var i:nat $\leftarrow$ 0 $\mathcal{O}(1)$
3: $res \leftarrow true$ $\mathcal{O}(1)$
4: **while** $i <$ LONGITUD(clv) $\wedge$ $res =_{\text{obs}} true$ **do** $\mathcal{O}(1)$
5: **if** $actual \rightarrow hijos[\text{ord}(clv[i])] =_{\text{obs}}$ NULL **then** $\mathcal{O}(1)$
6: $res \leftarrow false$ $\mathcal{O}(1)$
7: **else**$actual \leftarrow (actual \rightarrow hijos[\text{ord}(clv[i])])$ $\mathcal{O}(1)$
8: **end if**
9: **end while**
10: **if** $actual \rightarrow definicion =_{\text{obs}}$ NULL **then** $\mathcal{O}(1)$
11: $res \leftarrow false$ $\mathcal{O}(1)$
12: **end if**

Complejidad: |clv|

$\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1) + \sum_{i=1}^{|clv|}(\mathcal{O}(1) + max(\mathcal{O}(1), \mathcal{O}(1))) + \mathcal{O}(1) + max(\mathcal{O}(1), 0) =$
$4 * \mathcal{O}(1) + \sum_{i=1}^{|clv|}(\mathcal{O}(1) + \mathcal{O}(1)) + \mathcal{O}(1) =$
$5 * \mathcal{O}(1) + |clv| * 2 * \mathcal{O}(1) =$
$5 * \mathcal{O}(1) + 2 * \mathcal{O}(|clv|) =$
$2 * \mathcal{O}(|clv|) = \mathcal{O}(|clv|)$

iSignificado(**in** d: `diccString(`α`)`, **in** clv: `string`) $\rightarrow$ res : `diccString(`α`)`

1: var $actual$:puntero(e_nodo) $\leftarrow$ &(d) $\mathcal{O}(1)$
2: **for** var i:nat $\leftarrow$ 0 to LONGITUD(clv) **do** $\mathcal{O}(1)$
3: $actual \leftarrow (actual \rightarrow hijos[\text{ord}(clv[i])])$ $\mathcal{O}(1)$
4: **end for**
5: $res \leftarrow (actual \rightarrow definicion)$ $\mathcal{O}(1)$

Complejidad: |clv|

$\mathcal{O}(1) + \mathcal{O}(1) + \sum_{i=1}^{|clv|} \mathcal{O}(1) + \mathcal{O}(1) =$
$3 * \mathcal{O}(1) + |clv| * \mathcal{O}(1) =$
$3 * \mathcal{O}(1) + \mathcal{O}(|clv|) = \mathcal{O}(|clv|)$

5. Cola Prioritaria

5.1. TAD ColaPrioritaria

TAD ColaPrioritaria

igualdad observacional

$$(\forall c1, c2 : \text{colaP(Paquete)}) \left(c1 =_{\text{obs}} c2 \iff \begin{pmatrix} \text{vacía?}(c_1) =_{\text{obs}} \text{vacía?}(c_2) \wedge_{\text{L}} \\ (\neg\text{vacía}(c_1) \Rightarrow_{\text{L}} \\ (\text{próximo}(c_1) =_{\text{obs}} \text{próximo}(c_2) \wedge \\ \text{suCamino}(\text{próximo}(c_1)) \\ \text{suCamino}(\text{próximo}(c_2)) \wedge \\ \text{desencolar}(c_1) =_{\text{obs}} \text{desencolar}(c_2))) \end{pmatrix} =_{\text{obs}} \right)$$

géneros colaP(Paquete)

exporta colaP(Paquete), generadores, observadores

usa Bool, Nat, Paquete, Secu(α), Compu

observadores básicos

vacía?	: colaP(Paquete)	$\longrightarrow$	Bool	
próximo	: colaP(Paquete) *cp*	$\longrightarrow$	Paquete	{¬vacía?(cp)}
desencolar	: colaP(Paquete) *cp*	$\longrightarrow$	colaP(Paquete)	{¬vacía?(cp)}
suCamino	: Paquete p × colaP(Paquete) *cp*	$\longrightarrow$	Secu(Compu)	{está?(p, cp)}

generadores

vacía	:	$\longrightarrow$	colaP(Paquete)	
encolar	: Paquete p × colaP(Paquete) *cp*	$\longrightarrow$	colaP(Paquete)	{¬está?(p, cp)}
agCompu	: Paquete p × Compu c × colaP(Paquete) *cp*	$\longrightarrow$	colaP(Paquete)	{está?(p, cp)}

otras operaciones

está?	: Paquete p × colaP(Paquete) *cp*	$\longrightarrow$	Bool

axiomas $\forall p, p1, p2$: Paquete, $\forall c$: Compu, $\forall cp$: colaP(Paquete)

vacía?(vacía) ≡ true

vacía?(encolar(p, cp)) ≡ false

vacía?(agCompu(p, c, cp)) ≡ false

```
próximo(encolar(p, cp)) ≡ if vacía?(cp) then
                             p
                          else
                             if prioridad(p) < prioridad(próximo(cp)) then
                                p
                             else
                                próximo(cp)
                             fi
                          fi
próximo(agCompu(p, c, cp)) ≡ próximo(cp)
desencolar(encolar(p, cp)) ≡ if vacía?(cp) then
                                cp
                             else
                                if prioridad(p) < prioridad(próximo(cp)) then
                                   cp
                                else
                                   encolar(p, desencolar(cp))
                                fi
                             fi
desencolar(agCompu(p, c, cp)) ≡ desencolar(cp)
```

suCamino(p_1, encolar(p_2, cp)) ≡ **if** $p_1 = p_2$ **then** <> **else** suCamino(p_1, cp) **fi**

suCamino(p_1, agCompu(p_2, c, cp)) ≡ **if** $p_1 = p_2$ **then** suCamino(p_1, cp) ∘ c **else** suCamino(p_1, cp) **fi**

está(p, cp) $\equiv$ **if** vacía?(cp) **then**
false
else
if p = próximo(cp) **then** true **else** está?(p, desencolar(cp)) **fi**
fi

Fin TAD

Interfaz

se explica con: ColaPrioritaria.

géneros: `colaP(Paquete)`.

Operaciones básicas de COLA PRIORITARIA

Vacía?(**in** *cp*: `colaP(Paquete)`) $\rightarrow$ *res* : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacía?(*cp*)}
Complejidad: $\mathcal{O}(1)$
Descripción: verifica si una lista esta vacía

próximo(**in** *cp*: `colaP(Paquete)`) $\rightarrow$ *res* : `Paquete`
Pre $\equiv$ {$\neg$vacía?(*cp*)}
Post $\equiv$ {*res* $=_{\text{obs}}$ próximo(cp)}
Complejidad: $\mathcal{O}(log_2\ k)$

Desencolar(**in** *cp*: `colaP(Paquete)`) $\rightarrow$ *res* : `colaP(Paquete)`
Pre $\equiv$ {$c =_{\text{obs}} c_0 \wedge \neg$(vacia?(c))}
Post $\equiv$ {*res* $=_{\text{obs}}$ próximo(c_0) $\wedge$ $c =_{\text{obs}}$ desencolar(c_0)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el paquete mas prioritario.

Vacía() $\rightarrow$ *res* : `colaP(Paquete)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacía()}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea una nueva cola.

Agregar(**in/out** *c*: `colaP(Paquete)`, **in** *p*: `Paquete`)
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ encolar(p, c_0)}
Complejidad: $\mathcal{O}(log_2\ n)$ con n siendo la cantidad total de elementos en la cola.
Descripción: Agrega un paquete a la cola.

5.2. Representacion

Representación

Para representar la cola elegimos hacerla sobre un Heap. Sabiendo que los paquetes no están acotados, este Heap estará representado con nodos y punteros.

`colaP(Paquete)` **se representa con** `estr`

donde `estr` es `tupla`(*raiz*: `puntero`(nodo), *tam*: `nat`)
donde `nodo` es `tupla`(*paquete*: `Paquete`, *caminoReocrrido*: `Secu(Compu)`, *padre*: `puntero`(nodo), *hijoIzq*: `puntero`(nodo), *hijoDer*: `puntero`(nodo))

5.3. InvRep y Abs

CAMBIAR EL INVARIANTE POR EL DE HEAP!!

InvRep en lenguaje coloquial:

1. La componente "tam" de e_cola es igual a la cantidad de nodos en el arbol.
2. Todo nodo en el arbol tiene un unico padre, con excepcion de la raiz, que no tiene padre.
3. La relacion de orden es total.
4. Un nodo es mayor a otro si la componente "pri" del primero es mayor que la del segundo.
5. Un nodo es menor a otro si la componente "pri" del primero es menor que la del segundo.
6. No pueden haber dos nodos en el arbol que tengan el mismo numero en la componente "seg".
7. Si dos nodos tienen el mismo numero en la componente "pri", se procede a verficar la componente "seg" de ambos. El que tiene el mayor numero en dicha componente es el mayor, mientras que el otro es el menor.
8. Para cada nodo, todos los elementos del subarabol que se encuentra a la derecha de la raiz son mayores que la misma.
9. Para cada nodo, todos los elementos del subarabol que se encuentra a la izquierda de la raiz son menores que la misma.
10. La componente "alt" de cada nodo es igual a la cantidad de niveles que hay que recorrer para llegar a la hoja mas lejana.
11. Para cada nodo, la diferencia en modulo de la altura entre los dos subarboles del mismo no puede diferir en mas de 1.

CAMBIAR ABS POR LA DE HEAP!!
Abs:

Abs : colaP(Paquete) c $\longrightarrow$ colaP(Paquete) {Rep(c)}
Abs(c) $=_{\text{obs}}$ p: colaP(Paquete) | mismosProximos(c, p)

mismosProximos : $\langle$puntero(nodo) $\times$ nat$\rangle$ $\longrightarrow$ bool

mismosProximos(c, p) $\equiv$ **if** π_1(c) = NULL $\wedge$ vacia?(p) **then**
true
else
if (π_1(c) = NULL $\wedge$ $\neg$vacia?(p)) $\vee$ (π_1(c) $\neq$ NULL $\wedge$ vacia?(p)) **then**
false
else
if maxElem(*(π_1(c))) = proximo(p) **then**
mismosProximos(borrarMax(*(π_1(c)), borrar(π_1(proximo(p)), π_2(proximo(p)), p)
else
false
fi
fi
fi

Las funciones "maxElem" y "borrarMax" no han sido axiomatizadas. Ya que estamos trabajando con Arboles Binarios de Busqueda (en nuestro caso AVL) la logica de ambas funciones es la misma que esta expresada en el pseudocodigo del modulo. En particular "maxElem" se limita a buscar el nodo mas a la derecha del arbol, mientras que "borrarMax" una vez encontrado el maximo, procede a eliminarlo y reordenar el arbol.

5.4. Algoritmos

Algoritmos

IVACIA() → *res* : `colaP(Paquete)`

```
1: var res: colaP(Paquete) ← tupla(NULL, 0)                                   O(1)
```

Complejidad: $\mathcal{O}(1)$

IAGREGAR(**in/out** c: `colaP(Paquete)`, **in** p: `Paquete`)

```
1: if c.raiz == NULL then
2:     c.raiz ← &(tupla(p, <>, NULL, NULL, NULL))                             O(1)
3:     c.tam ← 1                                                              O(1)
4: else
5:     var iTam: int ← c.tam                                                  O(1)
6:     var iAux: int ← c.tam                                                  O(1)
7:     var aCoordenadas: arreglo[⌊log2(c.tam)⌋] de bool                       O(⌊log2(c.tam)⌋)
8:     while iTam > 0 do                                                      O(1)
9:         if iAux%2 == 0 then                                                O(1)
10:            aCoordenadas[iTam-1] = false                                   O(1)
11:        else
12:            aCoordenadas[iTam-1] = true                                    O(1)
13:        end if
14:        iAux ← ⌊iAux/2⌋                                                    O(1)
15:        iTam ← iTam - 1
16:    end while
17:    var seguir: bool ← true                                                O(1)
18:    var pNodo: puntero(nodo) ← c.raiz                                      O(1)
19:    var camino: arreglo[⌊log2(c.tam)⌋ + 1] de puntero(nodo)                O(⌊log2(c.tam)⌋ + 1)
20:    var nroCamino: nat
21:    camino[0] ← pNodo                                                      O(1)
22:    nroCamino ← 0                                                          O(1)
23:    while seguir == true do                                                O(1)
24:        if a ≥ *(pNodo).pri then                                           O(1)
25:            if a == *(pNodo).pri then                                      O(1)
26:                if b > *(pNodo).seg then                                   O(1)
27:                    if *(pNodo).der ! = NULL then                          O(1)
28:                        pNodo ← *(pNodo).der                               O(1)
29:                        nroCamino ← nroCamino + 1                          O(1)
30:                        camino[nroCamino] ← pNodo                          O(1)
31:                    else
32:                        *(pNodo).der ← &(tupla(a, b, pNodo, NULL, NULL, 1))  O(1)
33:                        nroCamino ← nroCamino + 1                          O(1)
34:                        camino[nroCamino] ← *(pNodo).der                   O(1)
35:                        seguir ← false                                     O(1)
36:                    end if
37:                else
38:                    if *(pNodo).izq ! = NULL then                          O(1)
39:                        pNodo ← *(pNodo).izq                               O(1)
40:                        nroCamino ← nroCamino + 1                          O(1)
41:                        camino[nroCamino] ← pNodo                          O(1)
42:                    else
43:                        *(pNodo).izq ← &(tupla(a, b, pNodo, NULL, NULL, 1))  O(1)
44:                        nroCamino ← nroCamino + 1                          O(1)
45:                        camino[nroCamino] ← *(pNodo).izq                   O(1)
46:                        seguir ← false                                     O(1)
47:                    end if
48:                end if
49:            else
50:                if *(pNodo).der ! = NULL then                              O(1)
```

```
51:                 pNodo ← *(pNodo).der                                                    O(1)
52:                 nroCamino ← nroCamino + 1                                               O(1)
53:                 camino[nroCamino] ← pNodo                                               O(1)
54:             else
55:                 *(pNodo).der ← &(tupla(a, b, pNodo, NULL, NULL, 1))                     O(1)
56:                 nroCamino ← nroCamino + 1                                               O(1)
57:                 camino[nroCamino] ← *(pNodo).der                                        O(1)
58:                 seguir ← false                                                          O(1)
59:             end if
60:         end if
61:     else
62:         if *(pNodo).izq ! = NULL then                                                   O(1)
63:             pNodo ← *(pNodo).izq                                                        O(1)
64:             nroCamino ← nroCamino + 1                                                   O(1)
65:             camino[nroCamino] ← pNodo                                                   O(1)
66:         else
67:             *(pNodo).izq ← &(tupla(a, b, pNodo, NULL, NULL, 1))                         O(1)
68:             nroCamino ← nroCamino + 1                                                   O(1)
69:             camino[nroCamino] ← *(pNodo).izq                                            O(1)
70:             seguir ← false                                                              O(1)
71:         end if
72:     end if
73: end while
74: c.tam ← c.tam + 1                                                                       O(1)
75: seguir ← true                                                                           O(1)
76: while nroCamino ≥ 0 ∧ seguir == true do                                                 O(⌊log2 N⌋ + 1)
77:     pNodo ← camino[nroCamino]                                                           O(1)
78:     *(pNodo).alt ← ALTURA(pNodo) v
79:     if |FACTORDESBALANCE(camino[nroCamino])| > 1 then                                   O(1)
80:         pNodo ← ROTAR(HIJOMASALTO (HIJOMASALTO(pNodo)), HijoMasAlto(pNodo), pNodo)      O(1)
81:         *(*(pNodo).izq).alt ← ALTURA(*(pNodo).izq)                                      O(1)
82:         *(*(pNodo).der).alt ← ALTURA(*(pNodo).der)                                      O(1)
83:         *(pNodo).alt ← ALTURA(*(pNodo))                                                 O(1)
84:         seguir ← false                                                                  O(1)
85:     end if
86:     nroCamino ← nroCamino − 1                                                           O(1)
87: end while
88: end if
```

Complejidad: $\mathcal{O}(1)$

Debido a la longitud del pseudocodigo, vamos a ignorar todas los condicionales y las asignaciones en la justificacion, ya que se realizan en tiempo constante. Solo nos vamos a centrar en dos puntos, la creacion del arreglo "camino" y el ultimo ciclo.
Para el arreglo asignamos esa cantidad de nodos ya que contamos con un Arbol balanceado, el cual como mucho puede necesitar de $\lfloor log_2\ N \rfloor + 1$ niveles para almacenar N nodos. Esta misma logica la utilizamos en el ultimo ciclo, en el cual para restaurar el balance del Arbol recorremos el mismo desde el ultimo nodo agregado (el cual es una hoja) hasta la raiz en el peor caso, corrigiendo cualquier desbalance en el camino.
Esto resulta en la siguiente suma:
$\mathcal{O}(\lfloor log_2\ N \rfloor + 1) + \mathcal{O}(\lfloor log_2\ N \rfloor + 1) =$
$2 * \mathcal{O}(\lfloor log_2\ N \rfloor + 1) =$
$\mathcal{O}(\lfloor log_2\ N \rfloor + 1) =$
$\mathcal{O}(\lfloor log_2\ N \rfloor) = \mathcal{O}(log_2\ N)$

IELIMINAR(**in/out** c: `colaP(Paquete)`, **in** a: `nat`, **in** b: `nat`)

```
1: var pNodo: puntero(nodo) ← c.raiz                                                        O(1)
2: ver seguir: bool ← true                                                                  O(1)
```

```
3: while pNodo ! = NULL ∧ seguir == true do                                  O(⌊log2 N⌋ + 1)
4:     if *(pNodo).pri == a ∧ *(pNodo).seg == b then                          O(1)
5:         seguir ← false                                                     O(1)
6:     else
7:         if a ≥ *(pNodo).pri then                                           O(1)
8:             if a == *(pNodo).pri then                                      O(1)
9:                 if b > *(pNodo).seg then                                   O(1)
10:                    pNodo ← *(pNodo).der                                   O(1)
11:                else
12:                    pNodo ← *(pNodo).izq                                   O(1)
13:                end if
14:            else
15:                pNodo ← *(pNodo).der                                       O(1)
16:            end if
17:        else
18:            pNodo ← *(pNodo).izq                                           O(1)
19:        end if
20:    end if
21: end while
22: if pNodo != NULL then                                                     O(1)
23:    bNodo: puntero(nodo) ← NULL                                            O(1)
24:    if pNodo == c.raiz then                                                O(1)
25:        if *(pNodo).izq == NULL ∧ *(pNodo).der == NULL then                O(1)
26:            c.raiz ← NULL                                                  O(1)
27:            delete pNodo                                                   O(1)
28:        end if
29:        if *(pNodo).izq ! = NULL ∧ *(pNodo).der == NULL then               O(1)
30:            c.raiz ← *(pNodo).izq                                          O(1)
31:            *(*(pNodo).izq).padre ← NULL                                   O(1)
32:            delete pNodo                                                   O(1)
33:        end if
34:        if *(pNodo).izq == NULL ∧ *(pNodo).der ! = NULL then               O(1)
35:            c.raiz ← *(pNodo).der                                          O(1)
36:            *(*(pNodo).der).padre ← NULL                                   O(1)
37:            delete pNodo                                                   O(1)
38:        end if
39:        if *(pNodo).izq ! = NULL ∧ *(pNodo).der ! = NULL then              O(1)
40:            var tNodo: puntero(nodo) ← pNodo.der                           O(1)
41:            while *(tNodo).izq != NULL do                                  O(⌊log2 N⌋ + 1)
42:                tNodo ← tNodo.izq                                          O(1)
43:            end while
44:            bNodo ← *(tNodo).padre                                         O(1)
45:            if *(tNodo).der ! = NULL then                                  O(1)
46:                *(*(tNodo).der).padre ← *(tNodo).padre                     O(1)
47:            end if
48:            if *(*(tNodo).padre).izq == tNodo then                         O(1)
49:                *(*(tNodo).padre).izq ← *(tNodo).der                       O(1)
50:            else
51:                *(*(tNodo).padre).der ← *(tNodo).der                       O(1)
52:            end if
53:            *(tNodo).padre ← *(pNodo).padre                                O(1)
54:            *(tNodo).izq ← *(pNodo).izq                                    O(1)
55:            *(tNodo).der ← *(pNodo).der                                    O(1)
56:            *(*(pNodo).izq).padre ← tNodo                                  O(1)
57:            *(*(pNodo).der).padre ← tNodo                                  O(1)
58:            delete pNodo                                                   O(1)
59:        end if
```

```
60:     else
61:         if *(pNodo).izq == NULL ∧ *(pNodo).der == NULL then                    O(1)
62:             if *(*(pNodo).padre).izq == pNodo then                             O(1)
63:                 *(*(pNodo).padre).izq ← NULL                                   O(1)
64:                 bNodo ← *(pNodo).padre                                         O(1)
65:                 delete pNodo                                                   O(1)
66:             else
67:                 *(*(pNodo).padre).der ← NULL                                   O(1)
68:                 bNodo ← *(pNodo).padre                                         O(1)
69:                 delete pNodo                                                   O(1)
70:             end if
71:         end if
72:         if *(pNodo).izq ! = NULL ∧ *(pNodo).der == NULL then                   O(1)
73:             if *(*(pNodo).padre).izq == pNodo then                             O(1)
74:                 *(*(pNodo).padre).izq ← *(pNodo).izq                           O(1)
75:                 *(*(pNodo).izq).padre ← *(pNodo).padre                         O(1)
76:                 bNodo ← *(pNodo).padre                                         O(1)
77:                 delete pNodo                                                   O(1)
78:             else
79:                 *(*(pNodo).padre).der ← *(pNodo).izq                           O(1)
80:                 *(*(pNodo).izq).padre ← *(pNodo).padre                         O(1)
81:                 bNodo ← *(pNodo).padre                                         O(1)
82:                 delete pNodo                                                   O(1)
83:             end if
84:         end if
85:         if *(pNodo).izq == NULL ∧ *(pNodo).der ! = NULL then                   O(1)
86:             if *(*(pNodo).padre).izq == pNodo then                             O(1)
87:                 *(*(pNodo).padre).izq ← *(pNodo).der                           O(1)
88:                 *(*(pNodo).der).padre ← *(pNodo).padre                         O(1)
89:                 bNodo ← *(pNodo).padre                                         O(1)
90:                 delete pNodo                                                   O(1)
91:             else
92:                 *(*(pNodo).padre).der ← *(pNodo).der                           O(1)
93:                 *(*(pNodo).der).padre ← *(pNodo).padre                         O(1)
94:                 bNodo ← *(pNodo).padre                                         O(1)
95:                 delete pNodo                                                   O(1)
96:             end if
97:         end if
98:         if *(pNodo).izq ! = NULL ∧ *(pNodo).der ! = NULL then                  O(1)
99:             var tNodo: puntero(nodo) ← pNodo.der                               O(1)
100:             while *(tNodo).izq != NULL do                                      O(⌊log2 N⌋ + 1)
101:                 tNodo ← tNodo.izq                                              O(1)
102:             end while
103:             bNodo ← *(tNodo).padre                                             O(1)
104:             if *(tNodo).der ! = NULL then                                      O(1)
105:                 *(*(tNodo).der).padre ← *(tNodo).padre                         O(1)
106:             end if
107:             if *(*(tNodo).padre).izq == tNodo then                             O(1)
108:                 *(*(tNodo).padre).izq ← *(tNodo).der                           O(1)
109:             else
110:                 *(*(tNodo).padre).der ← *(tNodo).der                           O(1)
111:             end if
112:             if *(*(pNodo).padre).izq == pNodo then                             O(1)
113:                 *(*(tNodo).padre).izq ← tNodo                                  O(1)
114:             else
115:                 *(*(tNodo).padre).der ← pNodo                                  O(1)
116:             end if
```

```
117:            *(tNodo).padre ← *(pNodo).padre                                  O(1)
118:            *(tNodo).izq ← *(pNodo).izq                                      O(1)
119:            *(tNodo).der ← *(pNodo).der                                      O(1)
120:            *(*(pNodo).izq).padre ← tNodo                                    O(1)
121:            *(*(pNodo).der).padre ← tNodo                                    O(1)
122:            delete pNodo                                                     O(1)
123:        end if
124:        c.tam ← c.tam + 1
125:        while b ! = NULL do                                                  O(⌊log2 N⌋ + 1)
126:            *(b).alt ← SetAltura(b)                                          O(1)
127:            if |FactorDeDesbalance(b)| > 1 then                              O(1)
128:                *(b).alt ← Rotar(HijoMasAlto (HijoMasAlto(b)), HijoMasAlto(b), b)   O(1)
129:                *(*(b).izq).alt ← SetAltura(*(b).izq)                        O(1)
130:                *(*(b).der).alt ← SetAltura(*(b).der)                        O(1)
131:                *(b).alt ← SetAltura(*(b))                                   O(1)
132:            end if
133:            b ← *(b).padre                                                   O(1)
134:        end while
135:    end if
136: end if
```

Complejidad: $\mathcal{O}(log_2\ N)$

Para la complejidad de este algoritmo nos vamos a remitir al mismo proceso que en el caso anterior, vamos a ignorar los condicionales y las asignaciones ya que estas se realizan en tiempo constante para centrarnos unicamente en los ciclos cuya complejidad depende de algun parametro.
En este algoritmo contamos con 4 ciclos que dependen de alguna variable, ninguno esta anidado con ningun otro ciclo, y en el peor caso solo recorremos 3 de ellos.
Esto nos da la siguiente suma:
$3 * \mathcal{O}(\lfloor log_2\ N \rfloor + 1) =$
$\mathcal{O}(\lfloor log_2\ N \rfloor + 1) =$
$\mathcal{O}(\lfloor log_2\ N \rfloor) = \mathcal{O}(log_2\ N)$

IRotar(**in/out** c: `colaP(Paquete)`, **in** $p1$: `puntero(nodo)`, **in** $p2$: `puntero(nodo)`, **in** $p3$: `puntero(nodo)`) → res : `puntero(nodo)`

```
 1: var t1: puntero(nodo) ← NULL                                                 O(1)
 2: var t2: puntero(nodo) ← NULL                                                 O(1)
 3: var t2: puntero(nodo) ← NULL v
 4: if (*(p3).pri ≤ *(p1).pri ∧ *(p3).seg < *(p1).seg) ∧
 5:    (*(p1).pri ≤ *(p2).pri ∧ *(p1).pri < *(p2).pri) then                      O(1)
 6:     t1 ← p3                                                                  O(1)
 7:     t2 ← p1                                                                  O(1)
 8:     t3 ← p2                                                                  O(1)
 9: end if
10: if (*(p3).pri ≥ *(p1).pri ∧ *(p3).seg > *(p1).seg) ∧
11:    (*(p1).pri ≥ *(p2).pri ∧ *(p1).pri > *(p2).pri) then                      O(1)
12:     t1 ← p2                                                                 O(1)
13:     t2 ← p1                                                                 O(1)
14:     t3 ← p3                                                                 O(1)
15: end if
16: if (*(p3).pri ≤ *(p2).pri ∧ *(p3).seg < *(p2).seg) ∧
17:    (*(p2).pri ≥ *(p1).pri ∧ *(p2).pri < *(p1).pri) then                      O(1)
18:     t1 ← p3                                                                 O(1)
19:     t2 ← p2                                                                 O(1)
20:     t3 ← p1                                                                 O(1)
21: end if
22: if (*(p3).pri ≥ *(p2).pri ∧ *(p3).seg > *(p2).seg) ∧
```

```
23:    (*(p2).pri ≥ *(p3).pri ∧ *(p2).pri > *(p3).pri) then                    O(1)
24:      t1 ← p1                                                               O(1)
25:      t2 ← p2                                                               O(1)
26:      t3 ← p3                                                               O(1)
27: end if
28: if c.raiz == p3 then
29:      c.raiz ← p3                                                           O(1)
30:      *(p3).padre ← NULL                                                    O(1)
31: else
32:      if *(*(p3).padre).izq = p3 then                                       O(1)
33:          CIZQ(*(p3).padre, t2)                                             O(1)
34:      else
35:          CDER(*(p3).padre, t2)                                             O(1)
36:      end if
37: end if
38: if *(t2).izq != p1 ∧ *(t2).izq != p2 ∧ *(t2).izq != p3 then
39:      CDER(t1, *(t2).izq)                                                   O(1)
40: end if
41: if *(t2).der != p1 ∧ *(t2).der != p2 ∧ *(t2).der != p3 then
42:      CDER(t3, *(t2).der)                                                   O(1)
43: end if
44: CIZQ(t2, t1)                                                               O(1)
45: CDER(t2, t3)                                                               O(1)
46: res ← t2                                                                   O(1)
```

Complejidad: $\mathcal{O}(1)$

Al igual que en los dos casos anteriores, debido a la longitud del pseudocodigo, vamos a ignorar los condicionales y las asignaciones ya que se realizan en tiempo constante.
Como todas las ejecuciones del codigo se efectuan en tiempo constante, podemos ver de manera trivial que la complejidad es $\mathcal{O}(1)$.

ICIZQ(**in** a : `puntero(nodo)`, **in** b : `puntero(nodo)`)

```
1: *(a).izq = b                                                                O(1)
2: *(b).padre = a                                                              O(1)
```

Complejidad: $\mathcal{O}(1)$

$\mathcal{O}(1) + \mathcal{O}(1) =$
$2 * \mathcal{O}(1) =$
$\mathcal{O}(1)$

ICDER(**in** a : `puntero(nodo)`, **in** b : `puntero(nodo)`)

```
1: *(a).der = b                                                                O(1)
2: *(b).padre = a                                                              O(1)
```

Complejidad: $\mathcal{O}(1)$

$\mathcal{O}(1) + \mathcal{O}(1) =$
$2 * \mathcal{O}(1) =$
$\mathcal{O}(1)$

ISETALTURA(**in** a : `puntero(nodo)`) $\rightarrow$ res : `nat`

```
1: if *(a).izq == NULL then                                                    O(1)
2:      if *(a).der == NULL then                                              O(1)
3:          res ← 1                                                           O(1)
4:      else
```

```
5:          res ← 1 + *(*(a).der).alt                                   O(1)
6:      end if
7:  else
8:      if *(a).der == NULL then                                        O(1)
9:          res ← 1 + *(*(a).izq).alt                                   O(1)
10:     else
11:         if *(*(a).izq).alt > *(*(a).der).alt then                   O(1)
12:             res ← 1 + *(*(a).izq).alt                               O(1)
13:         else
14:             res ← 1 + *(*(a).der).alt                               O(1)
15:         end if
16:     end if
17: end if
```

Complejidad: $\mathcal{O}(1)$

$\mathcal{O}(1) + max(\mathcal{O}(1) + max(\mathcal{O}(1), \mathcal{O}(1)), \mathcal{O}(1) + max(\mathcal{O}(1), \mathcal{O}(1) + max(\mathcal{O}(1), \mathcal{O}(1)))) =$
$\mathcal{O}(1) + max(\mathcal{O}(1) + max(\mathcal{O}(1), \mathcal{O}(1)), \mathcal{O}(1) + max(\mathcal{O}(1), \mathcal{O}(1) + \mathcal{O}(1))) =$
$\mathcal{O}(1) + max(\mathcal{O}(1) + \mathcal{O}(1), \mathcal{O}(1) + max(\mathcal{O}(1), 2 * \mathcal{O}(1))) =$
$\mathcal{O}(1) + max(2 * \mathcal{O}(1), 3 * \mathcal{O}(1)) =$
$\mathcal{O}(1) + 3 * \mathcal{O}(1) = 4 * \mathcal{O}(1) = \mathcal{O}(1)$

IFactorDesbalance(**in** a : `puntero(nodo)`) → res : `int`

```
1:  if *(a).izq == NULL then                                            O(1)
2:      if *(a).der == NULL then                                        O(1)
3:          res ← 0                                                     O(1)
4:      else
5:          res ← -(*(*(a).der).alt)                                    O(1)
6:      end if
7:  else
8:      if *(a).der == NULL then                                        O(1)
9:          res ← *(*(a).izq).alt                                       O(1)
10:     else
11:         res ← *(*(a).izq).alt - *(*(a).der).alt                     O(1)
12:     end if
13: end if
```

Complejidad: $\mathcal{O}(1)$

$\mathcal{O}(1) + max(\mathcal{O}(1) + (max(\mathcal{O}(1)), max(\mathcal{O}(1))), \mathcal{O}(1) + (max(\mathcal{O}(1)), max(\mathcal{O}(1)))) =$
$\mathcal{O}(1) + max(\mathcal{O}(1) + \mathcal{O}(1), \mathcal{O}(1) + \mathcal{O}(1)) =$
$\mathcal{O}(1) + max(2 * \mathcal{O}(1), 2 * \mathcal{O}(1)) =$
$\mathcal{O}(1) + 2 * \mathcal{O}(1) =$
$3 * \mathcal{O}(1) + \mathcal{O}(1)$

IHijoMasAlto(**in** a : `puntero(nodo)`) → res : `puntero(nodo)`

```
1:  if *(*(a).izq).alt >*(*(a).der).alt then                            O(1)
2:      res ← *(a).der                                                  O(1)
3:  else
4:      res ← *(a).izq                                                  O(1)
5:  end if
```

Complejidad: $\mathcal{O}(1)$

$\mathcal{O}(1) + max(\mathcal{O}(1), \mathcal{O}(1)) =$
$\mathcal{O}(1) + \mathcal{O}(1) =$
$2 * \mathcal{O}(1) = \mathcal{O}(1)$

IVacia?(**in** c: `colaP(Paquete)`) $\rightarrow$ res : `bool`

1: res $\leftarrow$ c.raiz ! = NULL $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$

IDesencolar(**in/out** c: `colaP(Paquete)`) $\rightarrow$ res : `tupla(a: nat, b: nat)`

1: var pNodo: puntero(nodo) $\leftarrow$ c.raiz $\mathcal{O}(1)$
2: **while** pNodo.der != NULL **do** $\mathcal{O}(log_2\ N)$
3: pNodo $\leftarrow$ pNodo.der $\mathcal{O}(1)$
4: **end while**
5: ELIMINAR(c, *(pNodo).pri, *(pNodo).seg) $\mathcal{O}(log_2\ N)$

Complejidad: $\mathcal{O}(log_2\ N)$

$\mathcal{O}(1) + \mathcal{O}(log_2\ N) * \mathcal{O}(1) + \mathcal{O}(log_2\ N) =$
$2 * \mathcal{O}(log_2\ N) = \mathcal{O}(log_2\ N)$

ITamaño(**in/out** c: `colaP(Paquete)`) $\rightarrow$ res : `nat`

1: res $\leftarrow$ c.tam $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$