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 **3**
1.1. Secu(α) 3
1.2. Mapa 3

2. Mapa **4**
2.1. Representaciorepresentacionn 4
2.2. InvRep y Abs 5
2.3. Algoritmos 5

3. DCNet **7**
3.1. Representacion 8
3.2. InvRep y Abs 8
3.3. Algoritmos 10

4. Diccionario String(α) **13**
4.1. Representacion 13
4.2. InvRep y Abs 14
4.3. Algoritmos 14

5. DiccRapido **16**
5.1. Representacion 16
5.2. InvRep y Abs 17
5.3. Algoritmos 17

1. Tad Extendidos

1.1. Secu(α)

otras operaciones

elemDeSecu : Secu(α) s $\times$ Nat n $\longrightarrow$ RUR {n $<$ 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 $\times$ estacion e_2 $\times$ Mapa m $\longrightarrow$ nat{$e_1, e_2 \subset$ estaciones(m) $\wedge_{\text{L}}$ 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 $\equiv$ $\{res =_{\text{obs}} \text{crearIt(computadoras}(r))\}$
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 $\equiv$ $\{\{c_1, c_2\} \subseteq \text{computadoras}(r)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{conectadas?}(r, c_1, c_2)\}$
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 $\equiv$ $\{\{c_1, c_2\} \subseteq \text{computadoras}(r) \wedge_{\text{L}} \text{conectadas?}(r, c_1, c_2)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{interfazUsada}(r, c_1, c_2)\}$
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 $\equiv$ $\{res =_{\text{obs}} \text{iniciarRed}()\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Crea una red sin computadoras.

AgregarComputadora(**in/out** r : `red`, **in** c : `compu`)
Pre $\equiv$ $\{r_0 =_{\text{obs}} r \wedge \neg(c \in \text{computadoras}(r))\}$
Post $\equiv$ $\{r =_{\text{obs}} \text{agregarComputadora}(r_0, c)\}$
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 $\equiv$ $\{r_0 =_{\text{obs}} \text{r} \wedge \{c_1, c_2\} \subseteq \text{computadoras}(r) \wedge \text{ip}(c_1) \neq \text{ip}(c_2) \wedge_{\text{L}} \neg\, \text{conectadas?}(r, c_1, c_2) \wedge \neg\, \text{usaInterfaz?}(r, c_1, i_1)$
$\wedge\, \neg\, \text{usaInterfaz?}(r, c_2, i_2)\}$
Post $\equiv$ $\{r =_{\text{obs}} \text{conectar}(r, c_1, i_1, c_2, i_2)\}$
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”.

$$
\begin{array}{lll}
\text{Rep} : \text{e\_mapa} \longrightarrow \text{bool} & & \\
\text{Rep}(m) \equiv \text{true} \Longleftrightarrow & & \\
& \text{m.estaciones} = \text{claves(m.uniones)} \wedge & 1. \\
& \text{m.\#sendas} = \text{\#sendasPorDos(m.estaciones, m.uniones)} / 2 \wedge \text{m.\#sendas} \leq \text{long(m.sendas)} \wedge_L & 2.\ 5. \\
& (\forall \text{ e1, e2: string})(\text{e1} \in \text{claves(m.uniones)} \wedge_L \text{e2} \in \text{claves(obtener(e1, m.uniones))} \Rightarrow_L & \\
& \text{e2} \in \text{claves(m.uniones)} \wedge_L \text{e1} \in \text{claves(obtener(e2, m.uniones))} \wedge_L & \\
& \text{obtener(e2, obtener(e1, m.uniones))} = \text{obtener(e1, obtener(e2, m.uniones))} \wedge & 3.\ 4. \\
& \text{obtener(e2, obtener(e1, m.uniones))} < \text{m.\#sendas}) \wedge & \\
& (\forall \text{ e1, e2, e3, e4: string})((\text{e1} \in \text{claves(m.uniones)} \wedge_L \text{e2} \in \text{claves(obtener(e1, m.uniones))} \wedge & \\
& \text{e3} \in \text{claves(m.uniones)} \wedge_L \text{e4} \in \text{claves(obtener(e3, m.uniones)))} \Rightarrow_L & \\
& (\text{obtener(e2, obtener(e1, m.uniones))} = \text{obtener(e4, obtener(e3, m.uniones))} \Longleftrightarrow & \\
& (\text{e1} = \text{e3} \wedge \text{e2} = \text{e4}) \vee (\text{e1} = \text{e4} \wedge \text{e2} = \text{e3})))) & 3.
\end{array}
$$

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

$$
\begin{array}{ll}
\#\text{sendasPorDos(c, d)} \equiv & \textbf{if } \emptyset?(\text{c}) \textbf{ then} \\
& \quad 0 \\
& \textbf{else} \\
& \quad \#\text{claves(obtener(dameUno(c),d))} + \#\text{sendasPorDos(sinUno(c), d)} \\
& \textbf{fi}
\end{array}
$$

$\text{Abs} : \text{e\_mapa}\ m \longrightarrow \text{mapa}$ $\{\text{Rep}(m)\}$

$$
\begin{array}{ll}
\text{Abs}(m) =_{\text{obs}} \text{p: mapa} \mid & \\
& \text{m.estaciones} = \text{estaciones(p)} \wedge_L \\
& (\forall \text{ e1, e2: string})((\text{e1} \in \text{estaciones(p)} \wedge \text{e2} \in \text{estaciones(p)}) \Rightarrow_L \\
& (\text{conectadas?(e1, e2, p)} \Longleftrightarrow \\
& \text{e1} \in \text{claves(m.uniones)} \wedge \text{e2} \in \text{claves(obtener(e2, m.uniones))))} \wedge_L \\
& (\forall \text{ e1, e2: string})((\text{e1} \in \text{estaciones(p)} \wedge \text{e2} \in \text{estaciones(p)}) \wedge_L \\
& \text{conectadas?(e1, e2, p)} \Rightarrow_L \\
& (\text{restriccion(e1, e2, p)} = \text{m.sendas[obtener(e2, obtener(e1, m.uniones))]} \wedge \text{nroConexion(e1,} \\
& \text{e2, m)} = \text{obtener(e2, obtener(e1, m.uniones)))} \wedge \text{long(restricciones(p))} = \text{m.\#sendas} \wedge_L (\forall \\
& \text{n:nat) (n} < \text{m.\#sendas} \Rightarrow_L \text{m.sendas[n]} = \text{ElemDeSecu(restricciones(p), n)))}
\end{array}
$$

2.3. Algoritmos

Algoritmos

iComputadoras(**in** r : `red`) → res : `itConj(Compu)`

$res \leftarrow$ 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`

$res \leftarrow$ 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`

$res \leftarrow$ 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`

$res \leftarrow$ 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`)

Agregar($r.computadoras$, c) $\mathcal{O}(1)$
Definir($r.vecinosEInterfaces$, c, Vacío()) $\mathcal{O}(|c|)$
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`)

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

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. DCNet

Interfaz

se explica con: DCNet, Iterador Unidireccional(α).

géneros: `dcnet`.

Operaciones básicas de DCNet

Red(**in** d: `dcnet`) $\rightarrow res$: `red`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{red}(d)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la red del dcnet.

CaminoRecorrido(**in** d: `dcnet`, **in** p: `paquete`) $\rightarrow res$: `secu(compu)`
Pre $\equiv \{p \in \text{paqueteEnTransito?}(d, p)\}$
Post $\equiv \{res =_{\text{obs}} \text{caminoRecorrido}(d, p))\}$
Complejidad: $\mathcal{O}(n * log_2(K))$
Descripción: Devuelve una secuencia con las computadoras por las que paso el paquete.

cantidadEnviados(**in** d: `dcnet`, **in** c: `compu`) $\rightarrow res$: `nat`
Pre $\equiv \{c \in \text{computadoras}(\text{red}(d))\}$
Post $\equiv \{res =_{\text{obs}} \text{cantidadEnviados}(d, c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la cantidad de paquetes que fueron enviados desde la computadora.

enEspera(**in** d: `dcnet`, **in** c: `compu`) $\rightarrow res$: `conj(paquete)`
Pre $\equiv \{c \in \text{computadoras}(\text{red}(d))\}$
Post $\equiv \{res =_{\text{obs}} \text{enEspera}(d, c)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve los paquetes que se encuentran en ese momento en la computadora.

IniciarDCNet(**in** r: `red`) $\rightarrow res$: `dcnet`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} \text{iniciarDCNet}(r)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Inicia un dcnet con la red y sin paquetes.

CrearPaquete(**in** p: `paquete`, **in/out** d: `dcnet`)
Pre $\equiv \{d_0 \equiv d \land \neg\, ((\exists\, p_1\colon \text{paquete})(\text{paqueteEnTransito}(s, p_1) \land \text{id}(p_1) = \text{id}(p)) \land \text{origen}(p) \in \text{computadoras}(\text{red}(d)) \land_{\text{L}}$ $\text{destino}(p) \in \text{computadoras}(\text{red}(d)) \land_{\text{L}} \text{hayCamino?}(\text{red}(d, \text{origen}(p), \text{destino}(p))\ \}$
Post $\equiv \{res =_{\text{obs}} \text{iniciarDCNet}(r)\}$
Complejidad: $\mathcal{O}()$
Descripción: Agrega el paquete al dcnet.

AvanzarSegundo(**in/out** d: `dcnet`))
Pre $\equiv \{d_0 \equiv d\ \}$
Post $\equiv \{d =_{\text{obs}} \text{avanzarSegundo}(c_0)\}$
Complejidad: $\mathcal{O}()$
Descripción: El paquete de mayor prioridad de cada computadora avanza a su proxima computadora siendo esta la del camino mas corto.

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`) $\to$ 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`) $\to$ 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`) $\to$ 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

`dcnet` se representa con `e_dc`

donde `e_dc` es `tupla`(*red*: `red`, *MasEnviante*: `tupla`(*compu*: `compu`, *enviados*: `nat`),
CompusYPaquetes: `DiccString`(*compu*: `compu`, `tupla`(*PorMasPrioritarios*:DiccRapido(
prioridad :nat, *PaquetitosdePrioridad*:conj(paquete)), *PaquetesYCaminos*:DiccRapido
(*paquetes*:paquete, *CaminoRecorrido*:secu(compu)), *Enviados*: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 $\iff$ claves(c.RURenEst) = estaciones(c.mapa) $\land$ 1
#RURHistoricos $\leq$ Long(c.RURs) $\land_L$ ($\forall$ i:Nat, t:<id:Nat, esta?:Bool, e:String, 2
inf:Nat, carac:Conj(Tag), sendEv: ad(Bool)>)
(i<#RURHistoricos $\land_L$ ElemDeSecu(c.RURs, i) = t $\Rightarrow_L$ (t.e $\in$ estaciones(c.mapa) 3
$\land$ t.id = i $\land$ tam(t.sendEv) = long(Restricciones(c.mapa)) $\land$
(t.inf >0 $\Rightarrow$ ($\exists$ j:Nat) (j <tam(t.sendEv) $\land_L$ $\neg$ (t.sendEv[j]))) $\land$
(t.esta? $\Leftrightarrow$ ($\exists$ e1: String) (e1 $\in$ claves(c.RUREnEst) $\land_L$ estaEnColaP?(obtener(e1, c.RUREnEst), t.id)))
$\land$ ($\forall$ h : Nat) (h <tam(t.sendEv) $\Rightarrow_L$
t.sendEv[h] = verifica?(t.carac, ElemDeSecu(Restricciones(c.mapa), h))))) $\land_L$
($\forall$ e1, e2: String)(e1 $\in$ claves(c.RUREnEst) $\land$ e2 $\in$ claves(c.RUREnEst) $\land$ e1 $\neq$ e2 $\Rightarrow_L$ 4
($\forall$ n:Nat)(estaEnColaP?(obtener(e1, c.RUREnEst), n) $\Rightarrow$ $\neg$ estaEnColaP?(obtener(e2, c.RUREnEst), n)
$\land$ n <#RURHistoricos $\land_L$ 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) $\land$ c.mapa = mapa(u) $\land_L$
robots(u) = RURQueEstan(c.RURs) $\land_L$
($\forall$ n:Nat) (n $\in$ robots(u) $\Rightarrow_L$ estacion(n,u) = c.RURs[n].e $\land$
tags(n,u) = c.RURs[n].carac $\land$ #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 $\land$ maxI = ciudad.#RURHistoricos
Abs : e_it u $\longrightarrow$ itUni(α) {Rep(u)}

Abs(u) $=_{\text{obs}}$ it: itUni(α) | (HayMas?(u) $\wedge_L$ Actual(u) = ciudad.RURs[it.i] $\wedge$ Siguientes(u, $\emptyset$) = VSiguientes(ciudad, it.i++, $\emptyset$) $\vee$ ($\neg$HayMas?(u))

Siguientes : itUniu × conj(RURs)cr ⟶ conj(RURs)

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

VSiguientes : ciudadc × Nati × conj(RURs)cr ⟶ conj(RURs)

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

3.3. Algoritmos

Algoritmos

iRed(**in** d: `dcnet`) → res : `red`

$res \leftarrow (d.red)$ $\mathcal{O}(1)$

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

iCaminoRecorrido(**in** d: `dcnet`,**in** p: `paquete`) → res : `secu(compu)`

var it ← computadoras(d.red) $\mathcal{O}(1)$
while HaySiguiente(it) **do** $\mathcal{O}(1)$
  if definido?(p.id,significado(Siguiente(it),CompusYPaquetes.d).PaquetesYCaminos) **then**
    res ← significado(p.id,significado(Siguiente(it),CompusYPaquetes.d).PaquetesYCaminos).CaminoRecorrido
  end if
  Avanzar(it) $\mathcal{O}(1)$
end while

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

iCantidadEnviados(**in** d: `dcnet`, **in** c: `compu`) → res : `nat`

res ← Significado(c,d.CompusYPaquetes).Enviados $\mathcal{O}(|c|)$

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

enEspera(**in** d: `dcnet`, **in** c: `compu`) → res : `itPaquete)`

res ← claves(Significado(c , d.CompusYPaquetes).PaquetesYCaminos) $\mathcal{O}(1)$

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

iIniciarDCNet(**in** r: `red`, **in/out** d: `dcnet`)

$\mathcal{O}(1)$

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

iCrearPaquete(**in** p : `rur`, **in/out** d : `dcnet`)

$\mathcal{O}(1)$

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

iAvanzarSegundo(**in/out** d : `dcnet`)

$\mathcal{O}(1)$

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

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

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

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

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`)

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

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`)

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

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: $itRURS \leftarrow$ 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) $=_{\text{obs}}$ n: diccString |
(∀ n:e_nodo) Abs(n) $=_{\text{obs}}$ 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 : string$s$ × e_nodo$n$ $\longrightarrow$ puntero(α)

obtenerDelArbol(s, n) ≡ **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** $actual \rightarrow hijos$[ord($clv[i]$)] $=_{\text{obs}}$ NULL **then** $\mathcal{O}(1)$
4: $actual \rightarrow (hijos$[ord($clv[i]$)] ← &(iNodoVacío())) $\mathcal{O}(1)$

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

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** $def : \alpha$) $\rightarrow res$: `bool`
var $actual$:puntero(e_nodo) $\leftarrow \&(d)$ $\mathcal{O}(1)$
var i:nat $\leftarrow 0$ $\mathcal{O}(1)$
$res \leftarrow true$ $\mathcal{O}(1)$
while $i <$ LONGITUD(clv) $\wedge\ res =_{\text{obs}} true$ **do** $\mathcal{O}(1)$
if $actual \rightarrow hijos[\text{ord}(clv[i])] =_{\text{obs}}$ NULL **then** $\mathcal{O}(1)$
$res \leftarrow false$ $\mathcal{O}(1)$
else$actual \leftarrow (actual \rightarrow hijos[\text{ord}(clv[i])])$ $\mathcal{O}(1)$
end if
end while
if $actual \rightarrow definicion =_{\text{obs}}$ NULL **then** $\mathcal{O}(1)$
$res \leftarrow false$ $\mathcal{O}(1)$
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`)`
var $actual$:puntero(e_nodo) $\leftarrow$ &(d) $\mathcal{O}(1)$
for var i:nat $\leftarrow 0$ to LONGITUD(clv) **do** $\mathcal{O}(1)$
$actual \leftarrow (actual \rightarrow hijos[\text{ord}(clv[i])])$ $\mathcal{O}(1)$
end for
$res \leftarrow (actual \rightarrow definicion)$ $\mathcal{O}(1)$

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. DiccRapido

Interfaz

se explica con: DICCIONARIO(CLAVE, SIGNIFICADO).

géneros: `diccRapido`.

Operaciones básicas de DICCRAPIDO

DEF?(**in** c: `clave`, **in** d: `diccRapido`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{def?}(c, d)\}$
Complejidad: $\mathcal{O}(log_2\ n)$, siendo n la cantidad de claves
Descripción: Verifica si una clave está definida.

OBTENER(**in** c: `clave`, **in** d: `diccRapido`) $\rightarrow$ res : `significado`
Pre $\equiv$ $\{\text{def?}(c, d)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{obtener}(c, d)\}$
Complejidad: $\mathcal{O}(log_2\ n)$, siendo n la cantidad de claves
Descripción: Devuelve el significado asociado a una clave

VACÍO() $\rightarrow$ res : `diccRapido`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{vacío}()\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Crea un nuevo diccionario vacío

DEFINIR(**in** c: `clave`, **in** s: `significado`, **in/out** d: `diccRapido`)
Pre $\equiv$ $\{d =_{\text{obs}} d_0\}$
Post $\equiv$ $\{d =_{\text{obs}} \text{definir}(c, s, d_0)\}$
Complejidad: $\mathcal{O}(log_2\ n)$, siendo n la cantidad de claves
Descripción: Define la clave, asociando su significado, al diccionario

BORRAR(**in** c: `clave`, **in/out** d: `diccRapido`)
Pre $\equiv$ $\{d =_{\text{obs}} d_0 \land \text{def?}(c, d_0)\}$
Post $\equiv$ $\{d =_{\text{obs}} \text{borrar}(c, d_0)\}$
Complejidad: $\mathcal{O}(log_2\ n)$, siendo n la cantidad de claves
Descripción: Borra la clave del diccionario

CLAVES(**in** d: `diccRapido`) $\rightarrow$ res : `itPaquete`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} \text{claves}(d)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve un iterador de paquete

5.1. Representacion

Representación

Para representar el diccionario, elegimos hacerlo sobre AVL. Sabiendo que la cantidad de claves no está acotada, este AVL estará representado con nodos y punteros. Cabe destacar, que las claves del diccionario deben contener una relación de orden.

`diccRapido` **se representa con** `estr`

donde `estr` es `tupla`(*raiz*: `puntero`(nodo), *tam*: `nat`)
donde `nodo` es `tupla`(*clave*: `clave`, *significado*: `significado`, *padre*: `puntero`(nodo), *izq*: `puntero`(nodo), *der*: `puntero`(nodo), *alt*: `nat`)

5.2. InvRep y Abs

5.3. Algoritmos

Algoritmos

IDef?(**in** c: `clave`, **in** d: `diccRapido`) $\rightarrow$ res : `bool`

var pNodo: puntero(nodo) $\leftarrow$ d.raiz $\mathcal{O}(1)$
while *(pNodo) != NULL **do** $\mathcal{O}(log_2\ n)$
    if *(pNodo).clave == c **then** $\mathcal{O}(1)$
        res $\leftarrow$ true $\mathcal{O}(1)$
        return res $\mathcal{O}(1)$
    else
        if c > *(pNodo).clave **then** $\mathcal{O}(1)$
            pNodo $\leftarrow$ *(pNodo).der $\mathcal{O}(1)$
        else
            pNodo $\leftarrow$ *(pNodo).izq $\mathcal{O}(1)$
        end if
    end if
end while
res $\leftarrow$ false $\mathcal{O}(1)$

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

IObtener(**in** c: `clave`, **in** d: `diccRapido`) $\rightarrow$ res : `significado`

var pNodo: puntero(nodo) $\leftarrow$ d.raiz $\mathcal{O}(1)$
while *(pNodo).clave != c **do** $\mathcal{O}(log_2\ n)$
    if c > *(pNodo).clave **then** $\mathcal{O}(1)$
        pNodo $\leftarrow$ *(pNodo).der $\mathcal{O}(1)$
    else
        pNodo $\leftarrow$ *(pNodo).izq $\mathcal{O}(1)$
    end if
end while
res $\leftarrow$ *(pNodo).significado $\mathcal{O}(1)$

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

IVacío() $\rightarrow$ res : `diccRapido`

var res: diccRapido $\leftarrow$ tupla(NULL, 0) $\mathcal{O}(1)$

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