Algoritmos y Estructura de Datos II

Primer Cuatrimestre de 2015

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

Reentrega Trabajo Práctico II

Grupo14

Integrante	LU	Correo electrónico
Ituarte Joaquín	457/13	joaquinituarte@hotmail.com
Ledezma Rocha Alexander	337/12	lralexandr@gmail.com
Zarate Eduardo Agustin	587/02	eazarate@yahoo.com.ar
Ignacio Gaston Viana Courtial	765/12	ivianacourtial@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Modulo Red(α)

Usa: VECTOR(α), CONJUNTO LINEAL(α), BOOL, NAT, STRING, TUPLA, COMPU .

Se explica con: RED

géneros: `red`.

Operaciones básicas de red.

Interfaz

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

AGREGARCOMPUTADORA(**in/out** r : `red`, **in** c : `compu`)
Pre $\equiv$ {$r =_{\text{obs}} r_0 \wedge \neg$ c $\in$ (computadoras(r))}
Post $\equiv$ {$r =_{\text{obs}} agregarComputadora(r_0, c)$}
Complejidad: $O(\#Computadoras(r) + long(c.IP))$
Descripción: Agrega una computadora a la red.
Aliasing: Agrega la computadora c por copia.

CONECTAR(**in/out** r : `red`, **in** c_1 : `compu`, **in** i_1 : `interfaz`, **in** c_2 : `compu`, **in** i_2 : `interfaz`)
Pre $\equiv$ {$r =_{\text{obs}} r_0 \wedge c_1 \in$ (computadoras(r)) $\wedge$ $c_2 \in$ (computadoras(r)) $\wedge$ $ip(c_1) \neq ip(c_2) \wedge \neg$ conectadas?(r, c_1, c_2) $\wedge \neg$ usaInterfaz$(r, c_1, i_1) \wedge \neg$ usaInterfaz(r, c_2, i_2)}
Post $\equiv$ {$r =_{\text{obs}} conectar(r_0, c_1, i_1, c_2, i_2)$}
Complejidad: $O(\#Computadoras(r) * max(long(c_1.IP), long(c_2.IP)))$
Descripción: Conecta las computadoras con las interfaces dadas.

COMPUTADORAS(**in** r : `red`) $\rightarrow$ *res* : `conj(compu)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} computadoras(r)$}
Complejidad: $O(1)$
Descripción: Devuelve el conjunto de computadoras de la red.
Aliasing: Devuelve una referencia al conjunto

CONECTADAS?(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {$c_1 \in$ Computadoras(r) $\wedge$ $c_2 \in$ Computadoras(r)}
Post $\equiv$ {$res =_{\text{obs}} Conectadas?(r, c_1, c_2)$}
Complejidad: $O(\#Computadoras(r) * max(long(c_1.IP), long(c_2.IP)))$
Descripción: Devuelve true si las computadoras estan conectadas entre si, en caso contrario false.

INTERFAZUSADA(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ *res* : `interfaz`
Pre $\equiv$ {$c_1 \in$ Computadoras(r) $\wedge$ $c_2 \in$ Computadoras(r) $\wedge$ Conectadas?(r, c_1, c_2)}
Post $\equiv$ {$res =_{\text{obs}} interfazUsada(r, c_1, c_2)$}
Complejidad: $O(\#Computadoras(r) * max(long(c_1.IP), long(c_2.IP)))$
Descripción: Devuelve la interfaz usada para conectar c1 con c2.
Aliasing: Devuelve una copia de la interfaz

VECINOS(**in** r : `red`, **in** c : `compu`) $\rightarrow$ *res* : `conj(compu)`
Pre $\equiv$ {c $\in$ computadoras(r)}
Post $\equiv$ {$res =_{\text{obs}} vecinos(r, c)$)}
Complejidad: $O(\#Computadoras(r)^2 * (max(long(c_1.IP), long(c_2.IP)) + \text{"}max.cantidaddeinterfacesdeunaCompu\text{"}))$
Descripción: Devuelve el conjunto de las computadoras que son vecinas de c.
Aliasing: Devuelve una copia del conjunto de computadoras.

UsaInterfaz?(**in** r : `red`, **in** c : `compu`, **in** i : `interfaz`) $\rightarrow$ res : `bool`
Pre $\equiv$ {c $\in$ computadoras(r)}
Post $\equiv$ $\{res =_{\text{obs}} usaInterfaz?(r, c, i)\}$
Complejidad: $O(\#Computadoras(r) * max(long(c_1.IP), long(c_2.IP)))$
Descripción: Devuelve true si la interfaz esta siendo usada, en caso contrario false.

Caminos(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `conj(lista(compu))`
Pre $\equiv$ $\{c_1 \in$ computadoras(r) $\wedge$ $c_2 \in$ computadoras(r)$\}$
Post $\equiv$ $\{res =_{\text{obs}} caminos(r, c_1, c_2)\}$
Complejidad: $O((L + Maximo\#Inter) * \#(Computadoras(r))! * \#(Computadoras(r))^2)$
Descripción: Devuelve el conjunto de caminos entre dos computadoras.
Aliasing: Se devuelven copias de los caminos.

HayCamino?(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{c_1 \in$ Computadoras(r) $\wedge$ $c_2 \in$ computadoras(r)$\}$
Post $\equiv$ $\{res =_{\text{obs}} hayCamino?(r, c_1, c_2)\}$
Complejidad: $O((L + Maximo\#Inter) * \#(Computadoras(r))! * \#(Computadoras(r))^2)$
Descripción: Devuelve true si hay camino entre las computadoras, false en caso contrario.

CaminosMinimos(**in** r : `red`, **in** c_1 : `compu`, **in** c_2 : `compu`) $\rightarrow$ res : `conj(lista(compu))`
Pre $\equiv$ $\{c_1 \in$ computadoras(r) $\wedge$ $c_2 \in$ computadoras(r)$\}$
Post $\equiv$ $\{res =_{\text{obs}} caminosMinimos(r, c_1, c_2)\}$
Complejidad: $O((L + Maximo\#Inter) * \#(Computadoras(r))! * \#(Computadoras(r))^2)$
Descripción: Devuelve el conjunto de los caminos minimos entre dos computadoras.
Aliasing: Se devuelven copias de los caminos.

Representación

Representación de Red

`red` se representa con `estr`

donde `estr` es `tupla`(*mapa*: `vector(vector(tupla(conecta?: bool, interfaz: nat)))` , *indexToString*: `vector(string)`, *computadoras*: `conj(compu)`)

Invariante de representacion en castellano:

1. La longitud de todos los vectores y subvectores es igual al cardinal del conjunto e.Computadoras.
2. Todas las strings de e.IndexToString son algún IP de alguna compu de e.Computadoras y no se repiten. Como longitud = cardinal, se que son los mismos entre vector y conjunto.
3. En la matriz (e.mapa), si un elemento e.mapa[i][j] posee el valor conecta? en true, entonces e.mapa[j][i].conecta? == true y las interfaces usadas son válidas y usadas una sola vez para tuplas con el valor.conecta? == true.

Rep : red $\longrightarrow$ bool

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

1. long(e.mapa) = long(e.indexToString)) $\wedge$
   long(e.mapa) = #e.computadoras $\wedge$
   ($\forall i$: nat) (i <long(e.mapa)) $\Rightarrow_{\text{L}}$ long(e.mapa) = long(e.mapa[i])
2. nombresValidos(e.indexToString, e.computadoras) $\wedge$
   long(sinRepetidos(e.indexToString)) = long(e.indexToString)
3. ($\forall i$: nat) (i <long(e.mapa)) $\Rightarrow_{\text{L}}$ interfacesValidas(e.mapa[i], obtenerInterfaces(e.indexToString[i], e.computadoras)) $\wedge$
   ($\forall i$: nat, j: nat) (i <long(e.mapa)) $\wedge$ (j <long(e.mapa)) $\Rightarrow_{\text{L}}$
   e.mapa[i][j].conecta? = e.mapa[j][i].conecta?)

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

Abs(e) $\equiv$ r: red | computadoras(r) = e.computadoras $\wedge$
($\forall$ c1,c2: Compu) (c1 $\neq$ c2 $\wedge$ $c_1 \in$ (computadoras(r)) $\wedge$ $c_2 \in$ (computadoras(r)) $\Rightarrow_{\text{L}}$ (conectadas?(r, c1, c2)
$\leftrightarrow$ e.mapa[ubicar(c1,e.indexToString)][ubicar(c2,e.indexToString)].conecta?
$\wedge$ ($\forall$ c1,c2: Compu) ($c_1 \in$ (computadoras(r)) $\wedge$ $c_2 \in$ (computadoras(r)) $\wedge$ conectadas?(r,c1,c2)) $\Rightarrow_{\text{L}}$
interfazUsada(r,c1,c2) = e.mapa[ubicar(c1,e.indexToString)][ubicar(c2,e.indexToString)].interfaz

Operaciones auxiliares de los TADS:

ubicar : string s $\times$ secu(string) sc $\longrightarrow$ nat {esta(sc, s)}

ubicar(s, sc) $\equiv$ **if** s = prim(sc) **then** 1 **else** 1 + ubicar(s, fin(sc)) **fi**

nombresValidos : secu(string) $\times$ conj(tupla(string $\times$ conj(nat))) $\longrightarrow$ bool

nombresValidos(s,c) $\equiv$ nombresValidos2(s, soloString(s))

nombresValidos2 : secu(string) $\times$ conj(string) $\longrightarrow$ bool

nombresValidos2(s, c) $\equiv$ **if** vacia(s) **then** true **else** prim(s) $\in$ c $\wedge$ nombresValidos(fin(s), c - {prim(s)}) **fi**

soloString : conj(tupla(string $\times$ conj(nat))) $\longrightarrow$ conj(string)

soloString(c) $\equiv$ **if** $\emptyset$?(c) **then** $\emptyset$ **else** Ag(Π_1(dameUno(c)),sinUno(c)) **fi**

interfacesValidas : secu(tupla(bool $\times$ nat)) $\times$ conj(nat) $\longrightarrow$ bool

interfacesValidas(s,c) $\equiv$ **if** vacia?(s) **then**
true
else
if Π_1(prim(s)) **then**
Π_2(prim(s)) $\in$ c $\wedge$ interfacesValidas(fin(s),c-(Π_1(prim(s))))
else
interfacesValidas(fin(s),c)
fi
fi

obtenerInterfaces : string $\times$ conj(tupla(string $\times$ conj(nat))) $\longrightarrow$ conj(nat)

obtenerInterfaces(s,c) $\equiv$ **if** $\emptyset$?(c) **then**
$\emptyset$
else
if s = Π_1(dameUno(c)) **then**
Π_2(dameUno(c))
else
obtenerInterfaces(s,sinUno(c))
fi
fi

Algoritmos

Algoritmos de Red

iniciarRed() $\rightarrow$ res: estr

begin
res.computadoras $\leftarrow$ vacio() //**O(1)**
res.mapa $\leftarrow$ vacia() //**O(1)**
res.indexToString $\leftarrow$ vacia() //**O(1)**
end
Data: Complejidad es O(1)

Algorithm 1: iniciarRed

```
agregarComputadora(inout e: estr, in c: compu)
begin
    AgregarRapido(e.computadoras, c)          //O(Longitud(IP(c)) + Cardinal(Interfaces(c)) )
    nat i ← 0                                                                    //O(1)
    while i <Longitud(e.mapa) do
        e.mapa[i] ← AgregarAtras(e.mapa[i], <false, 0>)                          //O(1)
        i++                                                                      //O(1)
    end
                                                                  //While: O(longitud(e.mapa))
    e.mapa ← AgregarAtras(e.mapa, vacia())                                       //O(1)
    i ← 0                                                                        //O(1)
    while (i <Longitud(e.mapa)) do
        AgregarAtras(ultimo(e.mapa), <false, 0>)                                 //O(1)
        i++
    end
                                                                //While: O(Longitud(e.mapa))
    e.indexToString ← AgregarAtras(e.indexToString, IP(c))     //Pasaje por referencia: O(1)
end
Data: Complejidad es O(Longitud(e.mapa) + Longitud(IP(c)))
```

Algorithm 2: agregarComputadora

```
conectar(inout e:estr, in c1:compu, in i1:interfaz, in c2:compu, in i2:interfaz)
begin
    nat pcindex1, pcindex2                                                       //O(1)
    nat i ← 0                                                                    //O(1)
    while i <Longitud(e.mapa) do
        if e.indexToString[i] == IP(c1)                  //O(Longitud(IP(c1))) then
            pcindex1 ← i                                                         //O(1)
        end
        if e.indexToString[i] == IP(c2)                  //O(Longitud(IP(c2))) then
            pcindex2 ← i                                                         //O(1)
        end
        i++                                                                      //O(1)
    end
                                                          //While: O(Longitud(e.mapa) * L)
    e.mapa[pcindex1][pcindex2] ← <true, interfaz1>                               //O(1)
    e.mapa[pcindex2][pcindex1] ← <true, interfaz2>                               //O(1)
end
Data: Complejidad es O(Longitud(e.mapa) * L)
```

Algorithm 3: conectar

```
Computadoras(in e:estr) → res: conj(compu)
begin
    res ← e.computadoras                                                         //O(1)
end
Data: Complejidad es O(1). Devuelve la referencia al conjunto
```

Algorithm 4: computadoras

conectadas?(in e:estr, in c1:compu, in c2:compu) ← res: bool

```
begin
    nat i,j                                                          //O(1)
    nat k ← 0                                                        //O(1)
    while k <Longitud(e.mapa) do
        if IP(c1) == e.indexToString[k]                              //O(L) then
            i ← k                                                    //O(1)
        end
        if IP(c2) = e.indexToString[k]                               //O(L) then
            j ← k                                                    //O(1)
        end
        k++                                                          //O(1)
    end
                                                                     //While: O(Longitud(e.mapa) * L)
    res ← e.mapa[i][j].conecta?                                      //O(1)
end
```

Data: Complejidad es O(Longitud(e.mapa) * L)

Algorithm 5: conectadas?

interfazUsada(in e:estr, in c1: compu, in c2: compu) ← res: nat

```
begin
    nat i,j                                                          //O(1)
    nat k ← 0                                                        //O(1)
    while k <Longitud(e.mapa) do
        if IP(c1) == e.indexToString[k]                              //O(L) then
            i ← k                                                    //O(1)
        end
        if IP(c2) == e.indexToString[k]                              //O(L) then
            j ← k                                                    //O(1)
        end
        k++                                                          //O(1)
    end
                                                                     //While: O(Longitud(e.mapa) * L)
    res ← e.mapa[i][j].interfaz                                      //Por copia. O(1)
end
```

Data: Complejidad es O(Longitud(e.mapa) * L). Devuelve una copia de la interfaz.

Algorithm 6: interfazUsada

vecinos(in e:estr, in c:compu) ← res: conj(compu)

```
begin
    res ← vacio()                                                    //O(1)
    itConj it ← crearIt(e.computadoras)                              //O(1)
    while haySiguiente(it) do
        if ¬(Siguiente(It) == c)    //O(L + "maxima Cantidad de Interfaces De Una Compu") then
            if conectadas?(e,c,siguienteIt(it))          //O(Cardinal(e.Computadoras) * L) then
                agregarRapido(res,siguiente(it))
                        //Agrega por copia: O(L + "maxima Cantidad de Interfaces De Una Compu")
            end
        end
        avanzar(it)                                                  //O(1)
    end
          //While: O(Cardinal(e.Computadoras)^2 * (L + "maxima Cantidad de Interfaces De Una
    Compu") )
end
```

Data: Complejidad es O(Cardinal(e.Computadoras)^2 * (L + "maxima Cantidad de Interfaces De Una Compu"))

Algorithm 7: vecinos

usaInterfaz?(in e:estr, in c:compu, in z:interfaz) → res: bool

```
begin
    nat j                                                                    //O(1)
    nat i ← 0                                                                //O(1)
    while i <Longitud(e.mapa) do
        if IP(c) == e.indexToString[i]                                       //O(L) then
            j ← i                                                            //O(1)
        end
        i++                                                                  //O(1)
    end
                                                                             //While: O(Longitud(e.mapa) * L)
    res ← false                                                              //O(1)
    i ← 0                                                                    //O(1)
    while i <longitud(e.mapa) do
        if e.mapa[j][i] == <true, z>                                         //O(1) then
            res ← true                                                       //O(1)
        end
        i++
    end
                                                                             //O(Longitud(e.mapa))
end
```

Data: Complejidad es O(Longitud(e.mapa) * L)

Algorithm 8: usaInterfaz?

caminosMinimos(in e: estr, in c1:compu, c2:compu) → res: conj(Lista(compu))

```
begin
    res ← vacio()                                                            //O(1)
    itConj it ← crearIt(caminos(e,c1,c2))
            //O( (L + Maximo#Inter) * Cardinal(e.Computadoras)! * Cardinal(e.Computadoras)^2)
    itConj it1 ← crearIt(caminos(e,c1,c2))
            //O( (L + Maximo#Inter) * Cardinal(e.Computadoras)! * Cardinal(e.Computadoras)^2)
    nat i ← 0                                                                //O(1)
    while HaySiguiente(it) do
        if i = 0 v Longitud(Siguiente(it)) <i then
            i ← Longitud(Siguiente(it))                                      //O(1)
        end
        avanzar(it)                                                          //O(1)
    end
                                                                             //While: O(Cardinal(e.Computadoras)!)
    while HaySiguiente(it1) do
        if i = 0 v Longitud(Siguiente(it1)) <i then
            agregarRapido(res, Siguiente(it1)))
                                    //Por copia: O(Cardinal(e.Computadoras) * (L + Maximo#Inter) )
        end
        Avanzar(it1)                                                         //O(1)
    end
    //While: O(Cardinal(e.Computadoras)! * Cardinal(e.Computadoras) * (L + Maximo#Inter))
end
```

Data: Complejidad es O((L + Maximo#Inter) * Cardinal(e.Computadoras)! * Cardinal(e.Computadoras)^2)

Algorithm 9: caminosMinimos

caminos2(in e: estr, in c1:compu, in c2:compu, inout l:lista(compu)) → res: conj(lista(compu))

```
begin
    agregarAtras(l,c1)                                  //Agrega por copia: O(L + Cardinal(Interfaces(c1)))
    res ← vacio()                                                                          //O(1)
    if c1 == c2                                                          //O(L + Maximo#Inter) then
        agregar(res,l)                          //Agrega por copia la lista: O(L + Maximo#Inter + Longitud(l))
    else
        conj(compu) vecinos ← vecinos(e,c1) //O(Cardinal(e.Computadoras)^2 * (L + Maximo#Inter) )
        itLista it ← creartIt(l)                                                           //O(1)
        while haySiguiente(it) do
            eliminar(vecinos,siguiente(it))                                                //O(1)
            avanzar(it)                                                                    //O(1)
        end
                                                                    //While: O(Cardinal(e.computadoras))
        itConjunto itVecinos ← crearIt(vecinos)                                            //O(1)
        res ← vacio()                                                                      //O(1)
        while haySiguiente(itVecinos) do
            conj(lista(compu)) caminosTemp ← caminos2(e,siguiente(itVecinos),c2,l)
            .                                   //Los caminos son solo aquellos que terminan en c2, si no es vacío.
            //O( (Cardinal(e.Computadoras)! * Cardinal(e.Computadora) * (L + Maximo#Inter) )
            itConjunto itCaminosTemp ← crearIt(caminosTemp)                                //O(1)
            while haySiguiente(itCaminosTemp) do
                agregar(res, siguiente(itCaminosTemp))
                                                //Por copia: O(Cardinal(e.computadoras)* (L + Maximo#Inter) )
                avanzar(itCaminosTemp)                                                     //O(1)
            end
                                    //O(While: Cardinal(e.Computadoras)! * Cardinal(e.Computadoras)^2 * (L +
            Maximo#Inter))
            avanzar(itVecinos)                                                             //O(1)
        end
                                                //While: O( (L + Maximo#Inter) * Cardinal(e.Computadoras)! *
        Cardinal(e.Computadoras)^2 )
    end
end
```

Data: Operación auxiliar de caminos. Los caminos que no terminen en c2 no terminan en el conjunto solución, dado que nunca se agregan por no entrar por la condición true del If. La recursión se realiza a lo sumo (Cardinal(e.Computadoras))! = n! veces, ya que empieza pudiendo hacer n-1 llamados y cada vez tiene un elemento menos en el cual realizar la recursión. Complejidad O(L^2 * Cardinal(e.Computadoras)! * Cardinal(e.Computadoras)^2)

Algorithm 10: caminos2

hayCaminos?(in e: estr, in c1:compu, c2:compu) → res: bool

```
begin
    res ← cardinal(caminosMinimos(e,c1,c2)) >0
          //O( (L + Maximo#Inter) * Cardinal(e.Computadoras)! * Cardinal(e.Computadoras)^2)
end
```

Data: Complejidad: O((L + Maximo#Inter) * Cardinal(e.Computadoras)! * Cardinal(e.Computadoras)^2)

Algorithm 11: hayCaminos?

caminos(in e:estr, in c1: compu, in c2: compu) → res:conj(lista(compu))

```
begin
    Lista(compu) l ← Vacía()                                                               //O(1)
    AgregarAtras(l, c1)                                                          //Por copia: O(L)
    res ← caminos2(e, c1, c2, l)          //O( (L + Maximo#Inter) * Cardinal(e.Computadoras)! *
    Cardinal(e.Computadoras)^2)
end
```

Data: Complejidad: O((L + Maximo#Inter) * Cardinal(e.Computadoras)! * Cardinal(e.Computadoras)^2)

Algorithm 12: caminos

2. Módulo DiccString(α)

Interfaz

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

Usa: BOOL, STRING, TUPLA

Se explica con: DICCIONARIO($String, \alpha$), ITERADOR UNIDIRECCIONAL(α).

géneros: `diccStr`(α), `itDiccStr`(α).

Operaciones básicas de DiccString(α)

VACÍO() $\rightarrow$ res : `diccStr`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} vacio$}
Complejidad: $O(1)$
Descripción: Genera un diccionario de strings vacío

DEFINIR(**in/out** ds : `diccStr`(α),**in** s : `String`, **in** d : α)
Pre $\equiv$ {$ds =_{\text{obs}} ds_0 \wedge$ s $\neq <>$}
Post $\equiv$ {$ds =_{\text{obs}} definir(ds0, s, d)$}
Complejidad: $O(|s| + copy(d))$
Descripción: Define la clave s con el significado d en el diccionario.
Aliasing: Copia la clave y el significado.

DEFINIDO?(**in** ds : `diccStr`(α),**in** s : `String`) $\rightarrow$ res : `bool`
Pre $\equiv$ {s $\neq <>$}
Post $\equiv$ {$res =_{\text{obs}} def?(ds, s)$}
Complejidad: $O(|s|)$
Descripción: Devuelve true si y sólo si s está definido en el diccionario.

SIGNIFICADO(**in** ds : `diccStr`(α),**in** s : `String`) $\rightarrow$ res : α
Pre $\equiv$ {def?(ds, s)}
Post $\equiv$ {$alias(res =_{\text{obs}} Significado(ds, s))$}
Complejidad: $O(|s|)$
Descripción: Devuelve el significado de la clave s en ds.
Aliasing: res es modificable si y sólo si ds es modificable.

Operaciones básicas del iterador(α)

CREARIT(**in** ds : `diccStr`(α)) $\rightarrow$ res : `itDiccStr`(α)
Pre $\equiv$ {true}
Post $\equiv$ {alias(res $=_{\text{obs}}$ crearItUni(significados(Diccionario(ds))) $\wedge_{\text{L}}$ res.actual $=_{\text{obs}}$ ds $\wedge$ vacia?(res.proximos))}
Complejidad: $O(1)$
Descripción: Crea un iterador unidireccional del diccionario.
Aliasing: El iterador se invalida si se modifica el diccionario original, res no es modificable

HAYMAS?(**in** $itds$: `itDiccStr`(α)) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} hayMas?(itds)$}
Complejidad: $O(1)$
Descripción: Devuelve true si y solo si en el iterador todavía quedan elementos para avanzar

ACTUAL(**in** *itds*: `itDiccStr(`α`)`) $\rightarrow$ *res* : `tupla(clave: String, significado:`α`)`
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ {alias($res =_{\text{obs}} actual(itds)$)}
Complejidad: $O(|max\_clave|)$
Descripción: Devuelve el elemento actual del iterador
Aliasing: El iterador se invalida si se modifica el diccionario original, res no es modificable

AVANZAR(**in/out** *itds*: `itDiccStr(`α`)`)
Pre $\equiv$ {it = $it_0 \wedge$ HayMas?(itds)}
Post $\equiv$ {$it =_{\text{obs}} Avanzar(it_0)$}
Complejidad: $O(|max\_clave|)$
Descripción: Avanza a la posición siguiente del iterador

Representación

Representación de DiccString

`diccStr(`α`)` **se representa con** `dstr`

donde dstr es puntero(nodo)

donde `nodo` es `tupla(`*letra*: `char` , *dato*: `puntero(`α`)`, *sigLetra*: `puntero(nodo)`, *restoPalabra*: `puntero(nodo)`)

Invariante de representacion en castellano:

1. El orden de los nodos con respecto a sigLetra debe ser creciente en base a Ord()
2. Si un nodo no tiene restoPalabra, se asegura que ahí hay un dato
3. Si sigLetra o restoPalabra no son null, se deberá cumplir el invariante en ellos tambien respectivamente.
4. No se generan ciclos.
5. Un nodo no puede tener dos padres.

Rep : dstr $\longrightarrow$ bool
Rep(ds) $\equiv$ true $\iff$ (ds $\neq$ NULL) $\Rightarrow_{\text{L}}$ Rep(*ds)

Rep : nodo $\longrightarrow$ bool
Rep(n) $\equiv$ true $\iff$

1. (n.sigLetra $\neq$ NULL) $\Rightarrow_{\text{L}}$ (ord(n.letra) $<$ ord(n.sigLetra$\rightarrow$letra) $\wedge$
2. (n.restoPalabra = NULL) $\Rightarrow_{\text{L}}$ n.dato $\neq$ NULL $\wedge$
3. (n.sigLetra $\neq$ NULL) $\Rightarrow_{\text{L}}$ Rep(n.sigLetra) $\wedge$ (n.restoPalabra = NULL) $\Rightarrow_{\text{L}}$ Rep(n.restoPalabra)
4. –
5. –

Abs : dstr ds $\longrightarrow$ dicc(String,α) {Rep(ds)}
Abs(ds) $\equiv$ Diccionario(ds, $<>$)

Diccionario : puntero(nodo) × string $\longrightarrow$ dicc(String,α)

Diccionario(nodo, s) $\equiv$ **if** nodo == NULL **then**
vacio
else
Fusionar(
if nodo.dato == NULL **then**
vacio
else
definir(s • (nodo→letra), nodo→dato, vacio)
fi,
Fusionar(Diccionario(nodo→restoPalabra, s • (nodo→letra)),Diccionario(nodo→sigLetra, s)))
fi

Fusionar : dicc(String × α) × dicc(String × α) $\longrightarrow$ dicc(String,α)

Fusionar(d,definir(k,s,d') $\equiv$ Fusionar(definir(k,s,d),d')

Fusionar(d,vacio) $\equiv$ d

Se llama a Fusionar(Caso1,Fusionar(Caso2,Caso3))

Caso1 abarca cuando se encuentra un dato definido
Caso2 es la recursion sobre el resto de la palabra (Por ejemplo, descubrimos que Caza esta definido, pero quizas esta Cazad definido tambien)
Caso3 es la recursion sobre el resto del diccionario (Por ejemplo, cuando llegamos al nodo Cazad, en sigLetra puede estar la r, que potencialmente indica que la clave Cazar esta definida)

Representación del iterador

`itDiccStr`(α) **se representa con** `itdstr`

donde `itdstr` es `tupla`(*actual*: `puntero(nodo)` , *proximos*: `Pila(puntero(nodo))`)

Donde nodo es el mismo definido anteriormente en el diccionario

Rep : itDiccStr(α) $\longrightarrow$ bool

Rep(it) $\equiv$ true $\Longleftrightarrow$ Rep(it.actual) $\wedge$ Rep(it.proximos)

Rep : Pila(puntero(nodo)) $\longrightarrow$ bool

Rep(pit) $\equiv$ true $\Longleftrightarrow$ **if** Vacia?(pit) **then** true **else** Rep(tope(pit)) $\wedge$ Rep(desapilar(pit)) **fi**

Y que para todo puntero A y B distintos pertenecientes a apilar(it.proximos,it.actual) ni A referencia a B ni B referencia a A, ni sucesivas referencias de sigLetra o restoPalabra de A referencian a ninguna sucesiva referencia de B y viceversa.

Abs : itDiccStr(α) it $\longrightarrow$ itUni(α) {Rep(it)}

Abs(it) $\equiv$ crearItUni(significados(Fusionar(Diccionario(it.actual, <>), DiccionarioPila(it.proximos, <>))))

DiccionarioPila : pila(diccString(string × α)) × string $\longrightarrow$ dicc(String,α)

DiccionarioPila(p,s) $\equiv$ Fusionar(Diccionario(tope(p),s), DiccionarioPila(desapilar(p),s))

Algoritmos Algoritmos de DiccString

iVacio() → res: DiccStr(α)

begin
| *res* ← NULL
end

Algorithm 13: iVacio

iSignificado(in ds: dstr, in s: string) → res: α

begin
| *res* ← iSignificadoAux(ds,s,0) O(|s| - 0) = O(|s|)
end
Data: Complejidad es O(|s|)

Algorithm 14: iSignificado

iSignificadoAux(in pn:puntero(nodo), in s:string, in n:nat) → res: α

begin
| **if** *pn → letra == s[n]* **then**
| | **if** *n == longitud(s) - 1* **then**
| | | res ← (pn → dato)
| | **else**
| | | res ← iSignificadoAux(pn → restoPalabra, s, n + 1)
| | **end**
| **else**
| | **if** *ord(pn → letra) <ord(s[n])* **then**
| | | res ← iSignificadoAux(pn → sigLetra, s, n)
| | **end**
| **end**
end
Data: Dado que la cantidad de letras distintas es acotada, recorrer horizontalmente la estructura tiene complejidad O(1). Por ende: Complejidad es O(|s| - n)

Algorithm 15: iSignificadoAux

iDefinido?(in ds:dstr, in s:string) → res: α

begin
| *res* ← *iDefinidoAux(ds,s,0)*
end
Data: Complejidad es O(|s|)

Algorithm 16: iDefinido?

iDefinir(inout ds:dstr, in s:string, in d: α)

begin
| iDefinirAux(ds, s, & copy(d), 0) O(|s| - 0) = O(|s|)
end
Data: Complejidad es O(|s|)

Algorithm 17: iDefinir

```
iDefinidoAux(in pn:puntero(nodo), in s:string, in n:nat) → res: bool
begin
    res ← false
    if pn != NULL then
        if pn → letra == s[n] then
            if n == longitud(s) - 1 then
                res ← (pn → dato) ≠ NULL
            else
                res ← iDefinidoAux(pn → restoPalabra, s, n + 1)
            end
        else
            if ord(pn → letra) <ord(s[n]) then
                res ← iDefinidoAux(pn → sigLetra, s, n)
            end
        end
    end
end
```

Data: Dado que la cantidad de letras distintas es acotada, recorrer horizontalmente la estructura tiene complejidad O(1). Por ende: Complejidad es O(|s| - n).

Algorithm 18: iDefinidoAux

```
iDefinirAux(inout pn:puntero(nodo), in s:string, in d:puntero(α), in n:nat)
begin
    if pn == NULL then
        pn ← crearDefinicion(s,d,n,pn)                                  O(|s| - n)
    else
        if s[n] == pn→letra then
            if n == longitud(s) - 1 then
                pn→dato ← d
            else
                iDefinirAux(pn→restoPalabra,s,d,n+1)                    O(|s| - n - 1)
            end
        else
            if Ord(s[n]) <Ord(pn→letra) then
                pn ← crearDefinicion(s,d,n,pn)                          O(|s| - n)
            else
                iDefinirAux(pn→sigLetra,s,d,n)                          O(|s| - n)
            end
        end
    end
end
```

Data: Dado que la cantidad de letras distintas es acotada, recorrer horizontalmente la estructura tiene complejidad O(1). Por ende: Complejidad es O(|s| - n).
Se itera recursivamente desde n inclusive hasta la longitud de s - 1 donde se corta la recursion y se devuelve el valor. Todas las operaciones menos crearDefinicion y los saltos recursivos son O(1). La recursion para encontrar la letra correspondiente en la lista de letras asociada con un nodo es O(1), si no se encuentra la letra correspondiente se crea una rama nueva con crearDefinicion, como esta funcion tiene la misma complejidad que una llamada recursiva, se mantiene la complejidad de O(|s| - n)

Algorithm 19: iDefinirAux

crearDefinicion(in s:string, in d:puntero(α), in n:nat, in pn:puntero(nodo)) → res: puntero(nodo)

```
begin
    res→letra ← s[n]
    res→sigLetra ← pn
    if n == longitud(s) - 1 then
        res→dato ← d
        res→restoPalabra ← NULL
    else
        res→dato ← NULL
        res→restoPalabra ← crearDefinicion(s, d, n+1, NULL)        O(|s| - n - 1)
    end
end
```

Data: Complejidad es O(|s| - n), se itera recursivamente desde n inclusive hasta la longitud de s - 1, donde se corta la recursión, todas las operaciones son O(1).

Algorithm 20: crearDefinicion

Algoritmos del Iterador

iCrearIt(in: ds: DiccStr(α)) → res: itDiccStr(α)

```
begin
    res.actual ← ds                                                O(1)
    res.proximos ← vacia()                                         O(1)
end
```

Data: Complejidad: O(1)

Algorithm 21: iCrearIt

iHayMas?(in: itds: itdstr) → res: bool

```
begin
    res ← actual ≠ NULL
end
```

Data: Complejidad: O(1)

Algorithm 22: iHayMas?

iActual(inOut: itds: itdstr) → res: α

```
begin
    if itds.actual→dato = NULL then
        iAvanzar(itds)
    end
    res = itds.actual→dato
end
```

Data: Complejidad es O(Max(claves(itds.actual)))

Algorithm 23: iActual

```
iAvanzar(inOut: itds: itdstr)
begin
    if itds.actual→sigLetra ≠ NULL then
        Apilar(itds.proximos, itds.actual→sigLetra)
    end
    if itds.actual→restoPalabra ≠ NULL then
        itds.actual ← itds.actual→restoPalabra
        if itds.actual→dato = NULL then
            iAvanzar(itds)
        end
    else
        if EsVacia?(itds.proximos) then
            itds.actual ← NULL
        else
            itds.actual ← Tope(itds.proximos)
            Desapilar(itds.proximos)
        end
    end
end
```

Data: El algoritmo busca el siguiente puntero con un dato no nulo. Buscar este dato tiene implica buscarlo horizontal (Complejidad O(1)) y verticalmente en el trie (Complejidad O(Longitud_Palabra)). La complejidad del algoritmo es O(|s|), donde s es la string más larga del diccionario.

Algorithm 24: iAvanzar

3. Módulo DCNet

Interfaz

se explica con: DCNet.

géneros: `DCNet`.

Usa: VECTOR(α), CONJUNTORAPIDO(α), RED, BOOL, COLAPRIOR(α), NAT, STRING, TUPLA, COMPU, PAQPRIOR, DICCSTRING, LISTA(COMPU).

Se explica con: DCNET

géneros: `DCnet`.

Operaciones básicas de DCNet

INICIARDCNET(**in** red: `red`) $\rightarrow$ res : `DCNet`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} IniciarDCNet(red)\}$
Complejidad: $\#O(Computadoras(r)^2 * (L + Maximo\#Inter) * \#(Computadoras(r))! * \#(Computadoras(r))^2)$
Descripción: Inicia una DCNet

CREARPAQUETE(**in/out** dc: `DCNet`,**in** p: `paquete`)
Pre $\equiv$ {$dc =_{\text{obs}} dc_0 \wedge$ paquete.origen $\in$ Computadoras(Red(dc)) $\wedge$ paquete.destino Computadoras(Red(dc)) $\wedge$ paquete.id $\notin$ Ids(dc) $\wedge_{\text{L}}$ HayCamino?(red(dc), paquete.origen, paquete.destino) }
Post $\equiv$ $\{dc =_{\text{obs}} crearPaquete(dc0, paquete.origen, paquete.destino)\}$
Complejidad: $O(L + log(k))$
Descripción: Agrega un paquete a la DCNet
Aliasing: Agrega por referencia.

AVANZARSEGUNDO(**in/out** dc: `DCNet`)
Pre $\equiv$ $\{dc =_{\text{obs}} dc_0\}$
Post $\equiv$ $\{dc =_{\text{obs}} avanzarSegundo(dc0)\}$
Complejidad: $O(n * (L + log(k)))$
Descripción: Se envian los paquetes de mayor prioridad de cada computadora en caso de tenerlos

RED(**in** dc: `DCNet`) $\rightarrow$ res : `Red`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} red(dc)\}$
Complejidad: $O(1)$
Descripción: Devuelve la red de la DCNet
Aliasing: Devuelve la referencia de la Red.

CAMINORECORRIDO(**in** dc: `DCNet`,**in** p: `paquete`) $\rightarrow$ res : `Lista(`$Compu$`)`
Pre $\equiv$ {PaqueteEnTransito?(dc, p)}
Post $\equiv$ $\{res =_{\text{obs}} CaminoRecorrido(dc, p)\}$
Complejidad: $O(n * log(k))$
Descripción: Devuelve la lista del camino recorrido por el paquete
Aliasing: Devuelve una referencia.

CANTIDADENVIADOS(**in** dc: `DCNet`,**in** pc: `Compu`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{pc \in Computadoras(red(dc))\}$
Post $\equiv$ $\{res =_{\text{obs}} CantidadEnviados(dc, pc)\}$
Complejidad: $O(L)$
Descripción: Devuelve la cantidad de envios que realizo la pc desde el inicio del DCNet

EnEspera(**in** dc: `DCNet`,**in** pc: `Compu`) $\rightarrow$ res : `conjR(paquetes)`
Pre $\equiv$ $\{pc \in Computadoras(red(dc))\}$
Post $\equiv$ $\{res =_{\text{obs}} EnEspera(dc, pc)\}$
Complejidad: $O(Longitud(IP(pc)))$
Descripción: Devuelve el conjunto de paquetes que tiene en espera la computadora.
Aliasing: Devuelve una referencia.

PaqueteEnTransito?(**in** dc: `DCNet`, **in** p: `paquete`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} PaqueteEnTransito?(dc, p)\}$
Complejidad: $O(n * log(k))$
Descripción: Devuelve si el paquete esta en la DCNet

LaQueMasEnvio(**in** dc: `DCNet`) $\rightarrow$ res : `Compu`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} LaQueMasEnvio(dc)\}$
Complejidad: $O(1)$
Descripción: Devuelve la computadora que más envios realizo
Aliasing: Devuelve una referencia.

Representación

Representación de DCNet

`DCNet` **se representa con** `estr`

donde `estr` es `tupla`(*Red*: `Red`, *Paquetes*: `Vector( DiccRapido(Paquete, Tupla(Camino: Lista(Compu), Posicion: nat)) )`, *ColaPC*: `Vector(colaPrior(PaqPrior))`, *Caminos*: `Vector(Vector(Vector(Compu)))`, *Contador*: `Vector(nat)` , *MasEnvios*: `Compu` , *IndexToPC*: `Vector(Compu)` , *PCToIndex*: `DiccString(nat)` , *EnEspera*: `conjR(paquete)`)

Rep en castellano:

1. Las longitudes de todos los vectores y subvectores coincide con el cardinal del conjunto de computadoras de la red.
2. Las computadoras de IndexToPC son las mismas que las del conjunto de computadoras de red (y como se que vale el rep de red y las longitudes son iguales, la cantidad de computadoras tambien es equivalente y no se repiten). Ademas, el índice correspondiente a cada compu de ese vector, es el mismo que el significado del diccionario PCToIndex usando dicha PC como clave.
3. Los paquetes son los mismos para cada índice en Paquetes y ColaPC
4. No se repiten los IDs de los paquetes en toda la DCNet.
5. El camino en dc.Paquete es algún camino válido de la Red, y la posición es un valor entre 0 y la longitud del camino.

Rep : estr $\longrightarrow$ bool

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

1. long(dc.Paquetes) = long(dc.ColaPC) $\wedge$ long(dc.ColaPC) = long(dc.Caminos) $\wedge$ long(dc.caminos) = long(dc.Contador) $\wedge$ long(dc.Contador) = long(dc.IndexToPC) $\wedge$ long(dc.EnEspera) = long(dc.caminos) $\wedge$ long(dc.IndexToPC) = #Computadoras(dc.Red) $\wedge$ #Computadoras(dc.red) = #Claves(dc.PCToIndex) $\wedge$
   ($\forall$i: nat | i <long(dc.ColaPC)) (long(dc.Caminos[i] = long(dc.Caminos) $\wedge$
2. ($\forall$i: nat | i <long(dc.ColaPC)) dc.IndextoPC[i] $\in$ Computadoras(dc.red) $\wedge$ i = Obtener(dc.PCToIndex, dc.IndexToPC[i].IP $\wedge$
3. ($\forall$i: nat | i <long(dc.ColaPC)) Claves(dc.Paquetes[i]) = Aconj(dc.ColaPC[i])))
4. Cardinal(Unir(dc.EnEspera)) = LongitudTotal(dc.EnEspera)
5. ($\forall$i: nat | i <long(dc.ColaPC)) ($\forall$p: Paquete | definido?(dc.Paquetes[i], p) Obtener(dc.Paquetes[i], p).camino $\in$ caminosMinimos(dc.Red)

LongitudTotal : Secu(conj(paquete)) $\longrightarrow$ nat

Aconj(sec) $\equiv$ **if** Vacia?(sec) **then** 0 **else** #(prim(sec)) + LongitudTotal(fin(sec)) **fi**

soloString : conj(paquete) $\longrightarrow$ conj(string)

soloString(c) $\equiv$ **if** $\emptyset$?(c) **then** $\emptyset$ **else** Ag(Π_1(dameUno(c)),sinUno(c)) **fi**

Unir : Secu(conj(Paquete)) $\longrightarrow$ Conj(String)

Unir(sec) $\equiv$ **if** Vacia?(sec) **then** $\emptyset$ **else** Ag(π1(prim(sec)), Unir(fin(sec))) **fi**

Abs : estr dc $\longrightarrow$ DCNet {Rep(dc)}

Abs(dc) $\equiv$ net: dcNet | red(net) = dc.red $\wedge$ ($\forall$ pc: Compu | pc $\in$ computadoras(red(net)) (CantidadEnviados(net, pc) = dc.contador[Obtener(dc.PCToIndex, pc.IP)] $\wedge$ EnEspera(net) = PasarAConj(dc.ColaPC[Obtener(dc.PCToIndex, pc.IP)])) $\wedge$ ($\forall$ p: Paquete | paqueteEntransito(net, p)) caminoRecorrido(net, p) = π1(Obtener(dc.Paquetes[Buscar(dc.Paquetes, p)]))

Buscar : Secu(Dicc(Paquete $\times$ Tupla(Secu(Compu)))) s $\times$ Paquete p $\longrightarrow$ nat {EstaPaquete(s, p)}

Buscar(sec, paq) $\equiv$ **if** π1(prim(sec)) = paq **then** 0 **else** 1 + Buscar(fin(sec), paq) **fi**

EstaPaquete : Secu(Dicc(Paquete $\times$ Tupla(Secu(Compu)))) s $\times$ Paquete p $\longrightarrow$ bool

EstaPaquete(sec, paq) $\equiv$ **if** vacia?(sec) **then**
False
else
if π1(prim(sec)) = paq **then** True **else** EstaPaquete(fin(sec), paq) **fi**
fi

Algoritmos Algoritmos de DCNet

Para los algoritmos, en cola de prioridad usamos el Tipo Paqprior para establecer el orden de prioridad entre las tuplas paquetes

iRed(in e: estr) $\rightarrow$ res: Red
begin
| $res \leftarrow$ e.Red //**O(1)**
end
Data: Complejidad: O(1)

Algorithm 25: red

iCaminoRecorrido(in e: estr, in p: Paquete) → res: Lista(Compu)

begin
- nat i ← 0 //**O(1)**
- nat n ← longitud(e.Paquetes) //**O(1)**
- **while** *i <n* **do**
  - **if** *ElemPertenece(e.Paquetes[i], p* ***//O(log(#Claves(e.Paquetes[i]))))*** **then**
    - res ← π1(Obtener(e.Paquetes[i], p)) //**El camino de la tupla <camino, posicion>**
    - //**O(1)**
  - **end**
  - i++ //**O(1)**
- **end**
- //**While: O(n*log(#Claves(e.Paquetes[i])))**

end

Data: Complejidad: O(n*(log(k)), donde k es la cantidad mayor de paquetes en una cola.

Algorithm 26: CaminoRecorrido

iCantidadEnviados(in e: estr, in pc: Compu) → res: nat

begin
- res ← e.Contador[Obtener(e.PCToIndex, IP(pc))] //**O(Longitud(IP(pc)))**

end

Data: Complejidad: O(Longitud(IP(pc)))

Algorithm 27: CantidadEnviados

iEnEspera(in e: estr, in pc: Compu) → res: conjR(Paquetes)

begin
- res ← e.EnEspera[Obtener(e.PCToIndex, pc.IP)] //**O(Longitud(IP(pc)))**

end

Data: Complejidad: O(L)

Algorithm 28: EnEspera

iPaqueteEnTransito(in e: estr, in p: Paquete) → res: bool

begin
- res ← False
- nat i ← 0 //**O(1)**
- nat k ← longitud(e.caminos) //**O(1)**
- **while** *(i<k)* **do**
  - **if** *ElemPertenece(e.Paquetes[i], p)* ***//O(log(#claves(e.Paquetes[i])))*** **then**
    - res ← True
  - **end**
  - i++ //**O(1)**
- **end**
- //**O(n*log(k))**

end

Data: Complejidad: O(n*log(k)), donde k es la cantidad mayor de paquetes en una cola.

Algorithm 29: PaqueteEnTransito

iLaQueMasEnvio(in e: estr) → res: nat

begin
- res ← e.MasEnvios

end

Data: Complejidad: O(1)

Algorithm 30: LaQueMasEnvio

iIniciarRed(in red: Red) → res: estr

begin
- res.Red ← Red //**O(1)**
- res.Paquetes ← Vacio() //**O(1)**
- res.ColaPC ← Vacio() //**O(1)**
- res.Caminos ← Vacio() //**O(1)**
- res.Contador ← Vacio() //**O(1)**
- res.IndexToPC ← Vacio() //**O(1)**
- res.PCToIndex ← CrearDicc() //**O(1)**
- res.EnEspera ← Vacio() //**O(1)**
- itConj(Compu) itConj ← CrearIt(computadoras(red)) //**O(1)**
- nat i ← 0 //**O(1)**
- **while** *HaySiguiente?(itConj)* **do**
  - AgregarAtras(res.Paquetes, VacioAVL()) //**O(Long(res.Paquetes))**
  - AgregarAtras(res.ColaPC, CrearCola()) //**O(longitud(e.ColaPC))**
  - AgregarAtras(res.Contador, 0) //**O(longitud(e.Contador))**
  - AgregarAtras(res.IndexToPC, siguiente(itConj)) //**O(Longitud(e.IndexToPC) + L)**
  - AgregarAtras(res.EnEspera, Vacio()) //**O(Longitud(e.EnEspera))**
  - Definir(res.PCToIndex(Siguiente(itConj).IP, i)) //**O(Longitud(IP(pc)))**
  - nat j ← 0 //**O(1)**
  - **while** *(j<i)* **do**
    - AgregarAtras(res.Caminos[j], Siguiente(CrearIt(CaminosMinimos(res.Red, IndexToPC[j])))
    - $//O((L + Maximo\#Inter) * \#(Computadoras(r))! * \#(Computadoras(r))^2)$
    - j++
  - **end**
  - //**While:** $O(i * (L + Maximo\#Inter) * \#(Computadoras(r))! * \#(Computadoras(r))^2)$
  - i++
- **end**
- //**While:** $\#O(Computadoras(r)^2 * (L + Maximo\#Inter) * \#(Computadoras(r))! * \#(Computadoras(r))^2)$
- res.MasEnvios ← res.IndexToPC[0] //**O(1)**

end

Data: Complejidad:
$\#O(Computadoras(r)^2 * (L + Maximo\#Inter) * \#(Computadoras(r))! * \#(Computadoras(r))^2))$

Algorithm 31: IniciarRed

iCrearPaquete(inout e: estr, in p: Paquete)

begin
- nat origen ← Obtener(e.PCToIndex, p.Origen.IP) //**O(L)**
- Definir(e.Paquetes[origen], p, <Vacia(), 0>) //**O(Log(#Claves(e.Paquetes[origen])))**
- Encolar(e.ColaPC[origen], NuevoPaqPrior(p)) //**O(log(Longitud(e.ColaPC)))**
- AgregarElem(e.EnEspera[origen], p) //**O(Cardinal(e.EnEspera[origen])**

end

Data: Complejidad: O(L + log(k)), donde k es la mayor cantidad de paquetes en una cola.

Algorithm 32: CrearPaquete

```
iAvanzarSegundo(inout e: estr)
begin
    nat i ← 0
    nat n ← Longitud(e.Caminos)                                                    //O(1)
    nat g ← 0                                                                      //O(1)
    while (i<n) do
        if !Vacio?(e.ColaPC[i])                                                    //O(1) then
            e.Contador[i]++                                                        //O(1)
            Paquete p ← Paquete(Desencolar(e.ColaPC[i]))                           //O(log(#Claves(e.ColaPC[i]))
            Compu actual ← e.IndexToPC[i]                                          //O(1)
            nat origen ← Obtener(e.PCToIndex(p.Origen.IP))                         //O(L)
            nat destino ← Obtener(e.PCToIndex(p.Destino.IP))                       //O(L)
            Tupla(Lista(Compu), nat) temp ← Obtener(p, e.Paquetes[i])              //O(log(#Claves(e.Paquetes[i])))
            BorrarElem(p, e.Paquetes[i])                                           //O(log(#Claves(e.Paquetes[i])))
            SacarElem(e.EnEspera[i], p)                                            //O(log(Cardinal(e.EnEspera[i])))
            if π2(temp) + 1 <Longitud(e.Caminos[origen][destino])                  //O(1) then
                Compu SiguientePC ← e.Caminos[origen][destino][π2(temp) +1]        //O(1)
                nat SigIndex ← Obtener(e.PCToIndex, SiguientePC.IP)                //O(L)
                Definir(e.Paquetes[SigIndex], p, Tupla(AgregarAtras(π1(temp), actual), π2(temp) +1) )
                                                                                   //O(log(#Claves(e.Paquetes[sigIndex]))
                Encolar(e.ColaPC[SigIndex], NuevoPaqPrior(p))                      //O(#Claves(e.ColaPC[SigIndex]))
                AgregarElem(e.EnEspera[SigIndex], p)                               //O(log(Cardinal(e.EnEspera[SigIndex])))
            else
            end
            i++                                                                    //O(1)
        end
        i ← 0                                                                      //O(1)
        nat max ← 0                                                                //O(1)
        while (i <n) do
            //Actualizo la pc que mas envios hizo
            if e.Contador[i] >e.Contador[max] then
                max ← i
            end
            i++                                                                    //O(1)
        end
                                                                                   //While: O(Longitud(e.contador))
        e.MasEnvios ← e.IndexToPC[max]                                             //O(1)
    end
end
```

Data: Complejidad: O(n*(L + log(k))), donde k es la mayor cantidad de paquetes de una cola, n es la cantidad de computadoras y L es la longitud maxima de un IP de compu

Algorithm 33: AvanzarSegundo

4. Módulo ColaDePrioridad(α)

Interfaz

parámetros formales
géneros α
funciones
MENOR(**in** a: α,**in** b: α) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ a <b}
Complejidad: $O(menor(a, b))$

se explica con: COLAPRIOR(), ITERADOR UNIDIRECCICONAL()..

géneros: `colaDePrior(), itCola()`..

Operaciones básicas de ColaDePrioridad(α)

VACIA() $\rightarrow$ res : `colaDePrior(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} vacia$}
Complejidad: $O(1)$
Descripción: Inicia una cola de prioridad vacia

VACIA?(**in** c: `colaDePrior(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} vacia?(c)$}
Complejidad: $O(1)$
Descripción: Devuelve si la cola de prioridad esta vacia o no

ENCOLAR(**in/out** c: `colaDePrior(`α`)`, **in** a: α)
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}} Encolar(c0, a)$}
Complejidad: $O(log(\#Claves(c)))$
Descripción: Encola el elemento a la cola
Aliasing: Agrega por copia si α es primitivo y por referencia en caso contrario.

DESENCOLAR(**in** c: `colaDePrior(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {$!vacia?(c) \wedge c = c_0$}
Post $\equiv$ {$res =_{\text{obs}} Proximo(c0) \wedge c =_{\text{obs}} desencolar(c0, a)$}
Complejidad: $O(log(\#Claves(c)))$
Descripción: Desencola y devuelve el elemento de mayor prioridad de la cola
Aliasing: Deuvelve una referencia.

Representación

Representación de colaDePrior(α)

`colaDePrior(`α`)` **se representa con** `estr`

donde `estr` es `tupla`(*raiz*: `puntero(nodo)`, *altura*: `nat`)
donde `nodo` es `tupla`(*valor*: α , *padre*: `puntero(nodo)` , *izq*: `puntero(nodo)` , *der*: `puntero(nodo)` , *#nodos*: `nat`)

Rep : estr $\longrightarrow$ bool

Rep($colaDePrior(\alpha)$) $\equiv$ true $\Longleftrightarrow$
Estr
raiz: Apunta al nodo que tiene el valor con mayor prioridad. Si la cola de prioridad es vacia, el puntero raiz es nulo. El nodo que es apuntado por raiz, no tiene padre.
altura: La altura del árbol (considerando el 0 el nivel de las hojas y el máximo el de la raíz).
Nodo
valor: Almacena un valor . Si tiene padre el nodo, el del nodo es menor al del padre.
izq: Puntero al nodo hijo izquierdo. No se pueden generar ciclos.
der: Puntero al nodo hijo derecho. No se pueden generar ciclos.
padre: Puntero al nodo padre. No se pueden generar ciclos.
#nodos: Consideramos al nodo actual como la raiz de un árbol. Esta variable representa la cantidad de nodos en ese árbol.
Además, el árbol que representa a la cola de prioridad está perfectamente balanceado y es izquierdista. Es decir, el último nivel esta lleno desde la izquierda.
No hay dos padres que apunten al mismo hijo.

Abs : estr c $\longrightarrow$ colaDePrior(α) {Rep(c)}

Abs(c) $\equiv$ IF (c.raiz = NULL) then vacia else EncolarSecu(Recorrer(c.raiz)) FI

Operaciones auxiliares:

EncolarSecu : secu(α) $\longrightarrow$ colaDePrior(α)

EncolarSecu(sec) $\equiv$ **if** vacia?(sec) **then** vacia **else** encolar(prim(sec), EncolarSecu(fin(sec))) **fi**

Recorrer : nodo $\longrightarrow$ secu(α)

Recorrer (nodo) $\equiv$ **if** nodo = NULL **then** <> **else** nodo.valor • Recorrer(nodo.izq) & Recorrer(nodo.der) **fi**

Algoritmos

Algoritmos de colaPrior(α)

iEncolar(inout c: estr, in a: α)

```
begin
    //Si α es no primitivo, el valor se agrega por referencia if c.raiz = NULL then
    | c.raiz ← & <a, NULL, NULL, NULL, 0 >                                  //O(1)
  else
    | puntero(nodo) n ← ultPoner(c.raíz, c.altura)                  //O(log(c.raiz.#nodos))
    | puntero(nodo) nuevo ← NULL                                            //O(1)
    | if n→izq = NULL then
    |   | n→izq ← & <a, NULL, NULL, NULL, 1 >                               //O(1)
    |   | nuevo ← n→izq                                                     //O(1)
    | else
    |   | n→der ← &<a, NULL, NULL, NULL, 1 >                                //O(1)
    |   | nuevo ← n→der                                                     //O(1)
    | end
  end
  | aumentar#Nodos(c, nuevo)                                      //O(log(c.raiz.#nodos))
  | subir (c, nuevo)                                             //O(log(c.raiz.#nodos))
  | c→altura ← calcularAltura(c)                                 //O(log(c.raiz.#nodos))
end
Data: Complejidad: O(log(c.raiz.#nodos))
```

Algorithm 34: Encolar

iDesencolar(in c: estr)α

begin
  res← c.raiz.valor //**O(1)**
  puntero(nodo) sacar = ultSacar(c) //**O(log(c.raiz.#nodos))**
  swap(sacar, c.raiz) //Sacar queda arriba de todo y c.raiz es una hoja //**O(1)**
  if *c.raiz→padre→izq = c.raiz* **then**
    c.raiz→padre→izq ← NULL //**O(1)**
  else
    c.raiz→padre→der ← NULL //**O(1)**
  end
  borrar(c.raiz) //los punteros del nodo, no el valor
  c.raiz ← sacar //**O(log(c.raiz.#nodos))**
  bajar(sacar) //**O(Log(c.raiz.#nodos))**
  //bajar no cambia los nodos, sino sus contenidos, por eso no debo actualizar la raiz de la cola
  calcularAltura(c) //**O(Log(c.raiz.#nodos))**
end
Data: Complejidad: O(log(c.raiz.#nodos))

Algorithm 35: Desencolar

iVacia()estr

begin
  res.raíz ← NULL //**O(1)**
  res.altura ← -1 //**O(1)**
end
Data: Complejidad: O(1)

Algorithm 36: Vacia

iVacia?(in c:estr)bool

begin
  res ← c.raiz == NULL //**O(1)**
end
Data: Complejidad: O(1)

Algorithm 37: Vacia?

iUltPoner(inout n: nodo, in i: nat) → puntero(nodo)

begin
  // Esta funcion no se exporta. . Al terminar la recursión, como recorre solamente una rama de nuestro arbol, su complejidad es de O(log(cantNodosCola))
  if *i==0* **then**
    res ← n //**O(1)**
  else
    nat cantNodosBalanceado ← $(2^{(k+1)}$ - 1)/2 //**O(1)**
    //El subarbol izq no esta lleno?
    if *cantNodosBalanceado >n→izq→ #nodos* **then**
      ultPoner(n→izq, i-1)
    else
      ultPoner(n→der, i-1)
    end
  end
end
Data: Busca el lugar donde se deberia colocar el siguiente nodo en la cola de prioridad, llamandose recursivamente solo sobre una rama del arbol. Complejidad: O(log(n.#nodos))

Algorithm 38: UltPoner

```
iUltSacar(inout n: nodo, in i: nat) → puntero(nodo)
begin
    // Esta funcion no se exporta. El valor i indica la altura del arbol.
    // El árbol es izquierdista, la recursión siempre recorre ramas para encontrar al nodo. Como las
    comparaciones son O(1) y recorrer una rama es log n, la complejidad es: log n (donde n es la cantidad de
    nodos del árbol)
    if i == 0 then
        res ← n
    else
        //El arbol esta lleno?
        if 2^(i+1) -1 == n→#nodos then
            res ← UltSacar(n→der, i-1)
        else
            //El subarbol izq no esta lleno?
            if 2^(i) -1 <n→izq→#nodos then
                res ← UltSacar(n→izq, i-1)
            else
                //Aca el derecho esta incompleto y quiero ver que tan incompleto esta
                if 2^(i-1) -1 == n→der→#nodos then
                    res← UltSacar(n→izq, i-1)
                else
                    res← UltSacar(n→der, i-1)
                end
            end
        end
    end
end
Data: Complejidad: O(log(n.#nodos))
```

Algorithm 39: UltSacar

```
iCalcularAltura(inout e: estr)
begin
    // Esta funcion no se exporta. 3:
    if e.raiz == NULL then
        e.altura ← -1                                                    //O(1)
    else
        puntero(nodo) n ← e.raiz                                         //O(1)
        while n != NULL do
            e.altura++                                                   //O(1)
            n ← n→izq                                                    //O(1)
        end
                                                                         //While: O(Log(#Claves(e)))
    end
end
Data: Calcula y modifica la altura del arbol de la cola de prioridad. Complejidad: O(Log(e.raiz.#nodos))
```

Algorithm 40: CalcularAltura

```
iSwap(inout n1: nodo, inout n2: nodo)
begin
    //Esta funcion no se exporta
    SwapEntorno(n1, n2)                                          //O(1)
    SwapEntorno(n2, n1)                                          //O(1)
    //Procedo a swapear los datos de los nodos
    puntero(nodo) temp ← n1→padre                                //O(1)
    n1→padre ← n2→padre                                          //O(1)
    n2→padre ← temp                                              //O(1)
    temp ← n1→izq                                                //O(1)
    n1→izq ← n2→izq                                              //O(1)
    n2→izq ← temp                                                //O(1)
    temp ← n1→der                                                //O(1)
    n1→der ← n2→der                                              //O(1)
    n2→der ← temp                                                //O(1)
    nat temporal ← n1→#nodos                                     //O(1)
    n1→#nodos ← n2rightarrow#nodos                               //O(1)
    n2→#nodos ← temporal                                         //O(1)
end
```

Data: swapea la informacion y el entorno de dos nodos. Complejidad O(1)

Algorithm 41: Swap

```
iSwapEntorno(inout n1: nodo, inout n2: nodo)
begin
    if n1→izq != NULL then
        n1→izq→padre ← n2                                        //O(1)
    end
    if n1→der != NULL then
        n1→der→padre ← n2                                        //O(1)
    end
    if n1→padre != NULL then
        if n1→padre→izq == n1 then
            n1→padre→izq ← n2                                    //O(1)
        end
        n1→padre→der ← n2                                        //O(1)
    end
end
```

Data: swapea el entorno de dos nodos. Complejidad O(1)

Algorithm 42: SwapEntorno

```
iAumentar#Nodos(inout n: nodo)
begin
    // Esta funcion no se exporta.
    puntero(nodo) temp ← n
    while n→padre != NULL do
        n ← n→padre
        n→#nodos++
    end
    While: //O(log(nodosDelArbol))
end
```

Data: aumenta la variable #nodos de un nodo y todos sus ascendientes en 1. Complejidad: O(log(nodosDelArbol))

Algorithm 43: Aumentar#nodos

```
iSubir(inout n: nodo)
begin
    // Esta funcion no se exporta.
    while n→padre != NULL ∧L Menor(n→padre→valor, n→valor) do
        Swap(n, n→padre)                                          //O(1)
        //n queda ahora como el padre, entonces no debo actualizar el puntero para el ciclo
    end
                                                                  //While: O(log(CantidadNodosArbol))
end
```

Data: Se utiliza para corregir la posicion de un nodo en el arbol de la cola de prioridad, sube el nodo de posicion de ser necesario. Complejidad: O(log(CantidadNodosArbol))

Algorithm 44: Subir

```
iBajar(inout n: nodo)
begin
    // Esta funcion no se exporta.
    if n→der != NULL then
        if Menor(n→izq→valor, n→der→valor) then
            Swap(n→der, n)
            Bajar(n)
        else
            Swap(n→izq, n)                                        //O(1)
            Bajar(n)
        end
    else
        if n→izq != NULL then
            Swap(n→izq, n)
            Bajar(n)
        end
    end
end
```

Data: Se utiliza para corregir la posicion de un nodo en el arbol de la cola de prioridad, baja el nodo de posicion de ser necesario. Complejidad: O(log(CantidadNodosArbol))

Algorithm 45: Bajar

5. Módulo PaqPrior

Interfaz

se explica con: TUPLA.

géneros: `PaqPrior`.

Operaciones básicas

NUEVOPAQPRIOR(**in** p: `paquete`) $\rightarrow$ res : `PaqPrior`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} Tupla(p, p.Prior)\}$
Complejidad: $O(1)$
Descripción: Crea un PaqPrior
Aliasing: Crea por referencia.

PAQUETE(**in** pp: `PaqPrior`) $\rightarrow$ res : `Paquete`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} pp.Paquete\}$
Complejidad: $O(1)$
Descripción: Devuelve el paquete dentro de pp
Aliasing: Devuelve por referencia

MENOR(**in** $pp1$: `PaqPrior`, **in** $pp2$: `PaqPrior`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} pp1.Prior < pp2.Prior\}$
Complejidad: $O(1)$
Descripción: Devuelve el si pp1 es menor a pp2

Representación

Representación de PaqPrior

`PaqPrior` **se representa con** `estr`

donde `estr` es `tupla`(*Paquete*: `paquete`, *Prior*: `nat`)

Rep : estr $\longrightarrow$ bool
Rep($PaqPrior$) $\equiv$ true $\Longleftrightarrow$ True

Abs : estr pp $\longrightarrow$ PaqPrior $\{\text{Rep}(pp)\}$
Abs(pp) $\equiv$ t: Tupla(Paquete, nat) | $\pi 1$(t) = pp.Paquete $\wedge$ $\pi 2$(t) = pp.Prior

Algoritmos Algoritmos de PaqPrior

```
iNuevoPaqPrior(in p: Paquete)estr
begin
    res.Paquete ← p
    res.Prior ← p.Prioridad
end
```

Algorithm 46: NuevoPaqPrior

iPaquete(in e: estr)Paquete

begin
| res ← e.Paquete
end

Algorithm 47: Paquete

iMenor(in e1: estr, in e2: estr)bool

begin
| res ← e1.Prior <e2.Prior
end

Algorithm 48: Menor

6. Modulo diccRapido(α, β)

parámetros formales
géneros α
función COPIAR(**in** a : α) $\rightarrow$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's
función $\bullet <\bullet$(**in** $a1$: α, **in** $a2$: α) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a1 < a2$}
Complejidad: $\Theta(a1 < a2))$
Descripción: Operador menor de α's

se explica con: DICCIONARIO(α, β).

géneros: `diccRapido`(α, β).

usa: BOOL, NAT TUPLA.

Operaciones Basicas de diccRapido

Interfaz

VACIO() $\rightarrow$ res : `diccRapido`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \emptyset$}
Complejidad: $O(1)$
Descripción: genera un conjunto vacio.

ELEMPERTENECE(**in** e : α, **in** c : `diccRapido`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} pertenece(e, c)$}
Complejidad: $O(Log(cantidad(c))$
Descripción: Verifica si un elemento esta en el conjunto.

BORRARELEM(**in** e : α, **in/out** c : `diccRapido`)
Pre $\equiv$ {$pertenece(e, c) \wedge c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}} claves(c_0)\$\{e\}$}
Complejidad: $O(Log(cantidad(c))$
Descripción: Elimina un elemento(pasado por parametro) del diccionario.
Aliasing: Borra la referencia del elemento.

DEFINIRELEM(**in** c : α,**in** s : β, **in/out** c : `diccRapido`)
Pre $\equiv$ {$c =_{\text{obs}} c_0$}
Post $\equiv$ {$c =_{\text{obs}} definir(c, s, c_0)$}
Complejidad: $O(Log(cantidad(c))$
Descripción: Agrega un elemento al diccionario
Aliasing: Define por referencia.

OBTENERELEM(**in** e : α, **in** c : `diccRapido`) $\rightarrow$ res : β
Pre $\equiv$ {$definido(e, c)$}
Post $\equiv$ {$res =_{\text{obs}} obtener(e, c)$}
Complejidad: $O(Log(cantidad(c))$
Descripción: Devuelve el significado del elemento pasado porparametro
Aliasing: Devuelve la referencia del elemento.

#DEFS(**in** c: `diccRapido`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} cardinal(c)$}
Complejidad: $cardinal(c)$
Descripción: Devuelve el numero de elementos definidos en el diccionario
Aliasing: Todos los parametros se pasan por referencia.

Representación

representacion de diccRapido

`diccRapido`(α, β) **se representa con** `str_diccRapido`

donde `str_diccRapido` es `tupla`(*raiz*: `str_nodo`)

donde `str_nodo` es `tupla`(*clave*: α , *significado*: β , *izq*: `str_diccRapido` , *der*: `str_diccRapido` , *padre*: `str_diccRapido` , *factorDeBalanceo*: `Nat` , *Altura*: `Nat`)

Inviariante de Representacion en castellano

1. El arbol puede ser vacio.
2. Todos los hijos a la izquierda son menores al padre.
3. Todos los hijos a la derecha son mas grandes que el padre.
4. Los subarboles tambien son avl's.
5. La diferencia de altura entre el hijo izquierdo y derecho
no es mayor a 1, es decir el factor de balanceo no es mayor a 1.

Rep : srt_diccRapido $\longrightarrow$ bool

Rep(c) $\equiv$ true $\iff$ esAvl?(c)

esAvl : estr_nodo $\longrightarrow$ bool

esAvl(*nodo*) $\equiv$ true $\iff$ (nodo == **Null**) $\vee_{\text{L}}$
(∀ c:α,(definido(c, nodo.izq)) $\iff$ c < (nodo.clave)) $\wedge$
(∀ c:α, (definido(c, nodo.der)) $\iff$ c > (nodo.clave)) $\wedge$
nodo.izq.Altura + 1 $\leq$ nodo.der.Altura $\vee$ nodo.izq.Altura - 1 $\leq$ nodo.der.Altura $\wedge$
esAvl(nodo.izq) $\wedge$ esAvl(nodo.der)

Inviariante de Abstraccion

Abs : estr c $\longrightarrow$ diccRapido {Rep(c)}

Abs(c) $\equiv$ dicc:str_diccRapido / (∀ m:α) definido(m,dicc) $=_{\text{obs}}$ elemPertenece(m,c) $\wedge$ obtener(m,dicc) $=_{\text{obs}}$ obtenerElem(m,dicc)

Algoritmos

iVacio () → res:stru_diccRapido
res.raiz ← Null
Data: Complejidad es O(1)

Algorithm 49: iVacio

iElemPertenece(in n:α, in e:str_diccRapido)→ res : Bool
begin
| puntero(nodo) nodoActual ← e.raiz O(1)
| **while** *nodoActual* $\neq$ *null* $\wedge$ $n \neq$ *nodoActual*→*clave* **do**
| | **if** $n >$ *nodoActual*→*clave* **then**
| | | nodoActual ← nodoActual→der O(1)
| | **else**
| | | nodoActual ← nodoActual→izq O(1)
| | **end**
| **end**
| res ← Igual(nodoActual→clave, n) O(1)
end
Data: Complejidad es O(Log(n)).

Algorithm 50: iElemPertenece

iBorrarElem(in e:str_diccRapido, in n:α)

begin **endData**:

puntero(nodo) nodoActual ← e.raiz	O(1)
puntero(nodo) padreActual ← Null	O(1)
while *nodoActual* ≠ *null* ∧ *n* ≠ *nodoActual*→*clave* **do**	
padreActual ← nodoActual	O(1)
if *n* > *nodoActual*→*clave* **then**	
nodoActual ← nodoActual→der	O(1)
else	
nodoActual ← nodoActual→izq	O(1)
end	
\\ complejidad del while: **(log(cardinal(e))** dado que en el peor de	
\\los casos tengo q bajar toda una rama del arbol, y eso es la altura del arbol	
\\ la estrategia seria buscar el antecesor mas proximo(o el sucesor),	
\\ una vez encontrado se swappean el contenido de nodoActual y el antecesor(o predecesor)	
\\ una hecho esto se elimina la hoja(si asi lo es) y voy rebalanceando el arbol hacia arriba	
end	
bool esHoja ← nodoActual→der==Null ∧ nodoActual→izq==Null	O(1)
\\ si no es hoja busco su pre o antecesor mas proximo	
puntero(nodo) nodoBuscador ← Null	O(1)
\\ nodoBuscador busca el pre/antecesor mas proximo	

```
\\ complejidad de el If O(log(n)+O(log(n)=2*O(log(n)=O(log(n)
\\ encontre el nodo con el que voy a swappear
nodo_clave tmpDato ← nodoActual→clave
nodoActual→clave = nodoBuscador→clave
nodoBuscador→clave = tmpDato
nodoActual = nodoBuscador
\\ nodoActual sigue teniendo el clave a borrar, y ahora es hoja
if esHoja then
    \\puede ser q haya sido hoja de una, y ahora tengo q ver si tiene padre.
    if padreActual ≠ Null then
        if padreActual ← der == nodoActual then
            padreActual← der = Null                                   O(1)
        else
            padreActual← izq = Null                                   O(1)
        end
    else
    end
else
    if padreActual→der == nodoActual then
        padreActual→der = nodoActual→der
        nodoActual → der → padre = padreActual
    else
        padreActual→izq = padreActual→izq
        nodoActual → izq → padre = padreActual
    end
end
if raiz == nodoActual then
    raiz = null
else
end
borrarNodo(nodoActual)
puntero(nodo) nodoDeArriba← padreActual
while nodoDeArriba ≠ Null
do
    if nodoDeArriba → izq ≠ null ∧ nodoDeArriba → der ≠ null then
        if nodoDeArriba → izq → altura < nodoDeArriba → der → altura then
            nodoDeArriba → altura = nodoDeArriba → der → altura
        else
            nodoDeArriba → altura = nodoDeArriba → izq → altura
        end
        nodoDeArriba →factorDeBalanceo = (nodoDeArriba → der → altura) − (nodoDeArriba → izq →
        altura)
    else
        if nodoDeArriba → izq ≠ Null then
            nodoDeArriba→altura = (nodoDeArriba→izq→ altura) + 1
            nodoDeArriba→factorDeBalanceo = −(nodoDeArriba→izq →altura)
        else
            if nodoDeArriba →der ≠ Null then
                nodoDeArriba→altura = (nodoDeArriba→der→ altura) + 1
                nodoDeArriba→factorDeBalanceo = (nodoDeArriba→der →altura)
            else
                nodoDeArriba→altura = 1
                nodoDeArriba→factorDeBalanceo = 0
            end
        end
    end
    nodoDeArriba ← nodoDeArriba→padre
end
```

```
nododeArriba ← padreActual
while nodoDeArriba ≠ Null do
    if nodoDeArriba→factorDeBalanceo ≠ 0 then
        if nodoDeArriba→factorDeBalanceo == 2 then
            if nodoDeArriba→der→factorDeBalanceo == -1 then
                rotacionDobleIzqAndDer(nodoDeArriba)
            else
                rotacionSimpleDerAndDer(nodoDeArriba)
            end
        else
        end
        if nodoDeArriba→factorDeBalanceo == -2 then
            if nodoDeArriba→izq→factorDeBalanceo ==1 then
                rotacionesDobleDerAndIzq(nodoDeArriba)
            else
                rotacionesSimpleIzqAndIzq(nodoDeArriba)
            end
        else
        end
    else
    end
end
\\ reajusto otra vez los factores de balanceo y altura de los nodo superiores \\el altorimo continua en la otra hoja nodoDeArriba ← nodoDeArriba→padre
if nodoDeArriba ≠ Null then
    if nodoDeArriba → izq ≠ null ∧ nodoDeArriba → der ≠ null then
        if nodoDeArriba → izq → altura < nodoDeArriba → der → altura then
            nodoDeArriba → altura = nodoDeArriba → der → altura
        else
            nodoDeArriba → altura = nodoDeArriba → izq → altura
        end
        nodoDeArriba →factorDeBalanceo = (nodoDeArriba → der → altura) − (nodoDeArriba → izq → altura)
    else
        if nodoDeArriba → izq ≠ Null then
            nodoDeArriba→altura = (nodoDeArriba→izq→ altura) + 1
            nodoDeArriba→factorDeBalanceo = −(nodoDeArriba→izq →altura)
        else
            if nodoDeArriba →der ≠ Null then
                nodoDeArriba→altura = (nodoDeArriba→der→ altura) + 1
                nodoDeArriba→factorDeBalanceo = (nodoDeArriba→der →altura)
            else
                nodoDeArriba→altura = 1
                nodoDeArriba→factorDeBalanceo = 0
            end
        end
    end
else
end
```

Data: Aqui finaliza borrarElemto. La complejidad es O(Log(n)) por los while's q se tiene(los while son 2*O(log(n))=O(log(n))), y el resto son asiganciones

Algorithm 51: iBorrarElem

iAgregarElem(in c: α, in s: β, in/out c:str_diccRapido)

```
puntero(nodo) padreActual← Null
puntero(nodo) nodoActual← c.raiz
\\buscamos el lugar para insertar el nuevo nodo
while nodoActual ≠ null ∧ c ≠ nodoActual→clave do
    padreActual ← nodoActual
    if c > noddoActual→ clave then
        nodoActual ← nodoActual→der
    else
        nodoActual ← nodoActual→izq
    end
end
\\ termine de buscar el lugar y ahora creo el nodo a insertar
puntero(Nodo) nuedoNuevo← nodoCrear(c,s)
\\lo insertamos...
if padreActual == Null then
    raiz ← nuevoNodo
else
    \\ aca es donde hacemos diferentes tipos de inserciones, primero la insercion a derecha
    if c > padreActual→clave then
        padreActual→der ← nuevoNodo
        nodoNuevo→padre ← padreActual
        padreActual→factorDeBalanceo ← (padreActual→factorDeBalanceo) + 1
    else
    end
    \\ insercion a izquierda
    if c < padreActual→clave then
        padreActual→izq ← nuevoNodo
        nodoNuevo→padre ← padreActual
        padreActual→factorDeBalanceo ← (padreActual→factorDeBalanceo) + 1
    else
    end
end
\\ reacomodo el factor de balanceo y la altura
If padreActual ≠ Null then
puntero(Nodo) nodoDeArriba ← padreActual
while nodoDeArriba ≠ Null
do
    if nodoDeArriba → izq ≠ null ∧ nodoDeArriba → der ≠ null then
        if nodoDeArriba → izq → altura < nodoDeArriba → der → altura then
            nodoDeArriba → altura = nodoDeArriba → der → altura
        else
            nodoDeArriba → altura = nodoDeArriba → izq → altura
        end
        nodoDeArriba →factorDeBalanceo = (nodoDeArriba → der → altura) − (nodoDeArriba → izq →
        altura)
    else
        if nodoDeArriba → izq ≠ Null then
            nodoDeArriba→altura = (nodoDeArriba→izq→ altura) + 1
            nodoDeArriba→factorDeBalanceo = −(nodoDeArriba→izq →altura)
        else
            if nodoDeArriba →der ≠ Null then
                nodoDeArriba→altura = (nodoDeArriba→der→ altura) + 1
                nodoDeArriba→factorDeBalanceo = (nodoDeArriba→der →altura)
            else
                nodoDeArriba→altura = 1
                nodoDeArriba→factorDeBalanceo = 0
            end
        end
    end
    nodoDeArriba ← nodoDeArriba→padre
end
```

```
\\ empezamos el rebalanceo
if padreActual ≠ Null then
    nodoDeArriba ← padreActual
    while nodoDeArriba ≠ Null do
        if nodoDeArriba→factorDeBalanceo ≠ 0 then
            if nodoDeArriba→factorDeBalanceo == 2 then
                if nodoDeArriba→der→factorDeBalanceo == -1 then
                    rotacionDobleIzqAndDer(nodoDeArriba)
                else
                    rotacionSimpleDerAndDer(nodoDeArriba)
                end
            else
            end
            if nodoDeArriba→factorDeBalanceo == -2 then
                if nodoDeArriba→izq→factorDeBalanceo ==1 then
                    rotacionDobleDerAndIzq(nodoDeArriba)
                else
                    rotacionSimpleIzqAndIzq(nodoDeArriba)
                end
            else
            end
        else
        end
    end
else
end
\\ volvemos reajustar los factores de rebalanceo y la altura
nodoDeArriba ← nodoDeArriba→padre
if nodoDeArriba ≠ Null then
    if nodoDeArriba → izq ≠ null ∧ nodoDeArriba → der ≠ null then
        if nodoDeArriba → izq → altura < nodoDeArriba → der → altura then
            nodoDeArriba → altura = nodoDeArriba → der → altura
        else
            nodoDeArriba → altura = nodoDeArriba → izq → altura
        end
        nodoDeArriba →factorDeBalanceo = (nodoDeArriba → der → altura) − (nodoDeArriba → izq →
        altura)
    else
        if nodoDeArriba → izq ≠ Null then
            nodoDeArriba→altura = (nodoDeArriba→izq→ altura) + 1
            nodoDeArriba→factorDeBalanceo = −(nodoDeArriba→izq →altura)
        else
            if nodoDeArriba →der ≠ Null then
                nodoDeArriba→altura = (nodoDeArriba→der→ altura) + 1
                nodoDeArriba→factorDeBalanceo = (nodoDeArriba→der →altura)
            else
                nodoDeArriba→altura = 1
                nodoDeArriba→factorDeBalanceo = 0
            end
        end
    end
else
end
```

Data: Aqui finaliza agregarElem. Complejidad es O(Log(n)) por lo while's que se tienen y elresto son asignaciones

Algorithm 52: iAgregarElem

iObtener(in e:str_diccRapido, in n:α)→ res : α

```
begin
    puntero(nodo) nodoActual ← e.raiz                                   O(1)
    while nodoActual ≠ null ∧ n ≠ nodoActual→clave do
        if n > nodoActual→clave then
            nodoActual ← nodoActual→der                                 O(1)
        else
            nodoActual ← nodoActual→izq                                 O(1)
        end
    end
    res ← nodoActual→Significado                                        O(1)
end
```

Data: Complejidad es O(Log(n)). El algoritmo busca en la estructura del dicc (un AVL) el nodo cuya clave sea igual a e . Si el elem e es mayor a la clave de la raiz busca en el sub-arbol izquierdo, caso contrario (el elemento e es menor) busca en sub-arbol derecho. Eventualmente llega la caso donde el elem e es igual a la clave del nodo y devolvemos el significado.
De esta forma evitamos compara cada uno de los nodos del arbol (Log(n)) y logramos optimizar la busqueda.

Algorithm 53: iObtener

i#Defs(in e:str_diccRapido)→ res : nat

```
begin
    puntero(nodo) nodoActual ← e.raiz                                   O(1)
    if nodoActual ≠ null then
        res ← res + i#Defs(nodoActual→der)i#Defs(nodoActual→izq)        O(n)
    else
        res ← 0
    end
end
```

Data: Complejidad es O(n). Se cuentan todos los elementos no nulos del dicc, uno por uno, de manera recursiva. De esta forma el algoritmo pasa por todos los nodos del la estructura (n) una vez.

Algorithm 54: i#Defs

Algoritmos Auxiliares

rotacionSimpleDerAndDer (in/out nodoAbalancear :puntero(Nodo))

```
puntero(nodo) padreSuperior ← nodoAbalancear→padre
puntero(nodo) F ← nodoAbalancear
puntero(nodo) d ← F→der
puntero(nodo) i ← d→izq
\\ burbujeamos nodo d por nodo F
if padreSuperior ≠ Null then
|  if padreSuperior→der == F then
|  |  padreSuperior→der ← d
|  else
|  |  padreSuperior→izq ← d
|  end
else
|  raiz ← d
end
\\ al hacer el burbujeo rompi cosas, por lo cual lo voy reconstruyendo...
F→der ← i
if i ≠ Null then
|  i→padre ← F
else
end
d→izq ← F
\\ ahora voy a reacomodar a los padres
F→padre ← d
d→padre ← padreSuperior
\\ ahora "esta" balanceado
if F→izq ≠ Null then
|  if i ≠ Null then
|  |  F→factorDeBalanceo ← (i→altura)−(F→izq→altura)
|  |  if i→altura > (F→izq→altura) then
|  |  |  (F→altura) ← i→altura
|  |  else
|  |  |  (F→altura) ← (F→izq→altura)+1
|  |  end
|  else
|  end
else
|  if i ≠ Null then
|  |  F→factorDeBalanceo ← (i→altura)
|  |  (F→altura) ← (i→altura)+1
|  else
|  |  F→factorDeBalanceo ← 0
|  |  F→altura ← 1
|  end
end
d→factorDeBalanceo ← (d→der→altura) − (F→altura
if F→altura > d→der→altura then
|  d→altura ← F→altura
else
|  d→altura ← (d→der→altura)+1
end
```

Data: complejidad **O(1)** son todas asignaciones

Algorithm 55: rotacionSimpleDerAndDer

```
rotacionSimmpleIzqAndIzq (in/out nodoAbalancear : puntero(Nodo))
puntero(nodo) padreSuperior ← nodoAbalancear→padre
puntero(nodo) F ← nodoAbalancear
puntero(nodo) i ← F→izq
puntero(nodo) d ← i→der
burbujeamos el nodo de i como padre de F
if padreSuperior ≠ Null then
|  if padreSuperior→der == F then
|  |  padreSuperior→der ← i
|  else
|  |  padreSuperior→izq ← i
|  end
else
|  raiz ← i
end
F→izq ← d
i→der ← F
F→padre ← i
i→padre ← padreSuperior
if d ≠ Null then
|  d→padre ← F
else
end
d→izq ← F
ahora es balanceado
if F→der ≠ Null then
|  if d ≠ Null then
|  |  F→factorDeBalanceo ← (F→izq→altura) − (d→altura)
|  |  if d→altura > (F→der→altura) then
|  |  |  (F→altura) ← d→altura
|  |  else
|  |  |  (F→altura) ← (F→der→altura)+1
|  |  end
|  else
|  |  F→factorDeBalanceo ← F→der→altura
|  |  (F→altura) ← (F→der→altura)+1
|  end
else
|  if d ≠ Null then
|  |  F→factorDeBalanceo ← −(d→altura)
|  |  (F→altura) ← (d→altura)+1
|  else
|  |  F→factorDeBalanceo ← 0
|  |  F→altura ← 1
|  end
end
i→factorDeBalanceo ← (F→altura − (i→izq→altura)
if F→altura > i→izq→altura then
|  i→altura ← F→altura
else
|  i→altura ← (i→izq→altura)+1
end
Data: complejidad O(1) son todas asignaciones
```

Algorithm 56: irotacionSimpleIzqAndIzq

```
rotacionDobleIzqAndDer (in/out nodoAbalancear : puntero(nodo))
puntero(nodo) padreSuperior ← nodoAbalancear→padre
puntero(nodo) F ← nodoAbalancear
puntero(nodo) d ← F→der
puntero(nodo) i ← d→izq
puntero(nodo) B ← i→izq
puntero(nodo) C ← i→der
if padreSuperior ≠ Null then
|   if padreSuperior→der == F then
|   |   padreSuperior→der ← i
|   else
|   |   padreSuperior→izq ← i
|   end
else
|   raiz ← i
end
i→padre ← padreSuperior
F→der ← B
if B ≠ Null then
|   B→padre ← F
else
end
d→izq ← C
if C ≠ Null then
|   C→padre ← d
else
end
i→der ← d
i→izq ← F
d→padre ← i
P→padre ← i
if F→izq ≠ Null then
|   if B ≠ Null then
|   |   F→factorDeBalanceo ← (B→altura) − (F→izq→altura)
|   |   if B→altura > (F→der→altura) then
|   |   |   (F→altura) ← B→altura
|   |   else
|   |   |   (F→altura) ← (F→der→altura)+1
|   |   end
|   else
|   |   F→factorDeBalanceo ← −(F→izq→altura)
|   |   (F→altura) ← (F→izq→altura)+1
|   end
else
|   if B ≠ Null then
|   |   F→factorDeBalanceo ← B→altura)
|   |   (F→altura) ← (B→altura)+1
|   else
|   |   F→factorDeBalanceo ← 0
|   |   F→altura ← 1
|   end
end
```

```
if d→der ≠ Null then
    if C ≠ Null then
        d→factorDeBalanceo ← (d→der→altura) − (d→altura)
        if C→altura > (d→der→altura) then
            (d→altura) ← C→altura
        else
            (d→altura) ← (d→der→altura)+1
        end
    else
        d→factorDeBalanceo ← (d→der→altura)
        (d→altura) ← (d→der→altura)+1
    end
else
    if C ≠ Null then
        d→factorDeBalanceo ← −(C→altura)
        (d→altura) ← (C→altura)+1
    else
        d→factorDeBalanceo ← 0
        d→altura ← 1
    end
end
if i→izq→altura > i→der→altura then
    i→altura ← i→izq→altura
else
    i→altura ← (i→der→altura)+1
end
i→factorDeBalanceo ← (i→der→altura) − (i→izq→altura)
```

Data: complejidad **O(1)** son todas asignaciones

Algorithm 57: irotacionDobleIzqAndDer

```
rotacionDobleDerAndIzq (in nodoDesbalanceado : puntero(nodo))
puntero(nodo) padreSuperior ← nodoAbalancear→padre
puntero(nodo) F ← nodoAbalancear
puntero(nodo) d ← F→izq
puntero(nodo) i ← d→der
puntero(nodo) B ← i→izq
puntero(nodo) C ← i→der
if padreSuperior ≠ Null then
    if padreSuperior→der == F then
        padreSuperior→der ← i
    else
        padreSuperior→izq ← i
    end
else
    raiz ← i
end
i→padre ← padreSuperior
F→izq ← C
if C ≠ Null then
    C→padre ← F
else
end
```

```
d→der ← B
if B ≠ Null then
|  B→padre ← d
else
end
i→der ← F
i→izq ← d
d→padre ← i
P→padre ← i
if F→der ≠ Null then
|  if C ≠ Null then
|  |  F→factorDeBalanceo ← (F→der→altura) − (C→altura)
|  |  if C→altura > (F→der→altura) then
|  |  |  (F→altura) ← C→altura
|  |  else
|  |  |  (F→altura) ← (F→der→altura)+1
|  |  end
|  else
|  |  F→factorDeBalanceo ← (F→der→altura)
|  |  (F→altura) ← (F→der→altura)+1
|  end
else
|  if C ≠ Null then
|  |  F→factorDeBalanceo ← −(C→altura)
|  |  (F→altura) ← (C→altura)+1
|  else
|  |  F→factorDeBalanceo ← 0
|  |  F→altura ← 1
|  end
end
if d→izq ≠ Null then
|  if B ≠ Null then
|  |  d→factorDeBalanceo ← (d→altura) − (d→izq→altura)
|  |  if B→altura > (d→izq→altura) then
|  |  |  (d→altura) ← B→altura
|  |  else
|  |  |  (d→altura) ← (d→izq→altura)+1
|  |  end
|  else
|  |  d→factorDeBalanceo ← −(d→izq→altura)
|  |  (d→altura) ← (d→izq→altura)+1
|  end
else
|  if B ≠ Null then
|  |  d→factorDeBalanceo ← (B→altura)
|  |  (d→altura) ← (B→altura)+1
|  else
|  |  d→factorDeBalanceo ← 0
|  |  d→altura ← 1
|  end
end
if i→izq→altura > i→der→altura then
|  i→altura ← i→izq→altura
else
|  i→altura ← (i→der→altura)+1
end
i→factorDeBalanceo ← (i→der→altura) − (i→izq→altura)
Data: complejidad O(1) son todas asignaciones
```

Algorithm 58: irotacionDobleDerAndIzq

iDameMaximo (in c : str_diccRapido) → elem:α
puntero(nodo) nodoActual ← c.raiz
puntero(nodo) maximo ← Null
while *nodoActual* ≠ *Null* **do**
  maximo ← nodoActual
  nodoActual ← (nodoActual→der
end
res ← maximo→clave
Data: Complejidad es **O(log(n)** por que recorre toda la altura del arbol

Algorithm 59: iDameMaximo

iCrearNodo (in c:α, in s:β → stru_nodo
nodoNuevo.clave ← c
nodoNuevo.significado ← s
nodoNuevo.padre ← Null
nodoNuevo.der ← Null
nodoNuevo.izq ← Null
nodoNuevo.altura ← 0
nodoNuevo.factorDeBalanceo ← 0

Observacion!
para usar este avl como queremos, vamos a extender el tad tupla con la operacion Menor

$[] < \_Tupla\ []$: p:Tupla × d:Tupla ⟶ bool

$< \_Tupla$ (p,d) ≡ **if** Π_1(p) < Π_1(d) **then** True **else** false **fi**

a la hora de meter tuplas en un avl, siepre se va meter esta comparacion, esto tambien serviria para comparar paquetes, ya tomaria el primer clave del paquete, osea el Id.

7. Modulo ConjuntoRapido(α)

parámetros formales
géneros α
función COPIAR(**in** a: α) $\rightarrow$ res : α
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a$}
Complejidad: $\Theta(copy(a))$
Descripción: función de copia de α's
función $\bullet < \bullet$(**in** $a1$: α, **in** $a2$: α) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} a1 < a2$}
Complejidad: $\Theta(a1 < a2))$
Descripción: Operador menor de α's

se explica con: CONJUNTO(α).

géneros: `conjR`(α).

usa: BOOL, DICCRAPIDO(α, BOOL)

Operaciones Basicas de conjR(α)

Interfaz

CONJVACIO() $\rightarrow$ res : `conjR`(α)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \emptyset$}
Complejidad: $O(1)$
Descripción: genera un conjunto vacio.

AGREGARELEM(**in** e: α, **in/out** c: `conjR`(α))
Pre $\equiv$ {true}
Post $\equiv$ {$c =_{\text{obs}} agregar(e, c)$}
Complejidad: $O(Log(\#(c))$
Descripción: agrega un elemento al conjunto
Aliasing: Agrega por referencia.

EPERTENECE(**in** e: α, **in** c: `conjR`(α)) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} e \in c$}
Complejidad: $O(Log(\#(c))$
Descripción: Verifica si un elemento esta en el conjunto.

SACARELEM(**in** e: α, **in/out** c: `conjR`(α))
Pre $\equiv$ {$e \in c$}
Post $\equiv$ {$c =_{\text{obs}} c - \{e\}$}
Complejidad: $O(Log(\#(c))$
Descripción: elimina un el elemento e del conjunto.
Aliasing: Elimina la referencia.

#ELEM(**in** c: `conjR`(α)) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \#(c)$}
Complejidad: $\#(c)$
Descripción: Devuelve el numero de elemtos del conjunto

Representación

`conjR`(α) **se representa con** `str_conjR`(α)

donde **str_conjR**(α) es **diccRapido(α, bool)**

Rep : srt_conjR(α) $\longrightarrow$ bool

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

Abs : srt_conjR(α) c $\longrightarrow$ conjunto(α) {Rep(c)}

Abs(c) $\equiv$ c: conjunto(α) |
($\#c =_{obs}$ #Elem(c)) $\wedge$
($\forall$a: α)(a$\in$ c $\Longleftrightarrow$ ePertenece(a, c)) $\wedge$
($\forall$a: α)(c - a $=_{obs}$ sacarElem(a, c)) $\wedge$
($\forall$a: α)(Ag(a,c) $=_{obs}$ agregarElem(a, c))

Algoritmos

iconjVacio () $\rightarrow$ res: str_conjR(α)
begin
| res $\leftarrow$ vacio() O(1)
end
Data: Complejidad: O(1). Ver complejidad de vacio()

iagregarElem(in e:α in/out co: str_conjR(α))
begin
| definirElem(e, true ,co) O(Log(#(c)))
end
Data: Complejidad: O(Log(#(c))). Ver complejidad de definirElem()

iePertenece(in e:α, co: str_conjR(α)) $\rightarrow$ res: bool
begin
| res $\leftarrow$ elemPertenece(e, co) O(Log(#(c)))
end
Data: Complejidad: O(Log(#(c))). Ver complejidad de elemPertenece()

isacarElem(in e:α in/out co: str_conjR(α))
begin
| borrarElem(e, co) O(Log(#(c)))
end
Data: Complejidad: O(Log(#(c))). Ver complejidad de borrarElem()

i#Elem(in co: str_conjR(α)) $\rightarrow$ res: nat
begin
| res $\leftarrow$ #Defs(e, co) O(#(c))
end
Data: Complejidad: O(#(c)). Ver complejidad de #Defs()

8. Módulo Compu

Interfaz

usa: String, nat, Conjunto Lineal(α).

se explica con: Compu.

géneros: `compu`.

Operaciones básicas

NuevaCompu(**in** IP: `String`, **in** $Inter$: `conj(nat)`) $\rightarrow$ res : `compu`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} Tupla(IP, Inter)\}$
Complejidad: $O(Long(IP) + \#Inter)$
Descripción: Crea una compu nueva.
Aliasing: Crea una pc por copia.

IP(**in** pc: `compu`) $\rightarrow$ res : `String`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} pc.IP\}$
Complejidad: $O(1)$
Descripción: Devuelve el IP de la compu.
Aliasing: Devuelve una referencia al IP.

Interfaces(**in** pc: `compu`) $\rightarrow$ res : `conj(nat)`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} pc.Interfaces\}$
Complejidad: $O(1)$
Descripción: Devuelve el conjunto de interfaces.
Aliasing: Devuelve una referencia al conjunto.

Copiar(**in** pc: `compu`) $\rightarrow$ res : `compu`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} pc\}$
Complejidad: $O(Long(IP(pc)) + \#Interfaces(pc))$
Descripción: Devuelve el conjunto de interfaces.
Aliasing: Devuelve una copia de la pc.

Representación

Representación de compu

`compu` **se representa con** `estr`

donde `estr` es `tupla`(IP: `String`, $Interfaces$: `conj(nat)`)

Rep : estr $\longrightarrow$ bool
Rep($compu$) $\equiv$ true $\Longleftrightarrow$ True

Abs : estr pc $\longrightarrow$ compu {Rep(pc)}
Abs(pc) $\equiv$ compu: Compu | compu.IP = pc.IP $\wedge$ compu.Interfaces = pc.Interfaces

Algoritmos **Algoritmos de compu**

iNuevaCompu (in IP: String, in Interfaces: conj(nat))estr

begin
| res.IP ← Copy(IP) //**O(Long(IP))**
| res.Interfaces ← Copy(Interfaces) //**O(Cardinal(Interfaces))**
end

Algorithm 60: NuevaCompu

iIP (in pc: estr)String

begin
| res ← pc.IP //**Referencia: O(1))**
end

Algorithm 61: IP

iInterfaces (in pc: estr)conj(nat)

begin
| res ← pc.Interfaces //**Referencia: O(1))**
end

Algorithm 62: Interfaces

iCopiar (in pc: estr)estr

begin
| res ← NuevaCompu(pc.IP, pc.Interfaces) **O(Long(pc.IP) + Cardinal(pc.Interfaces))**
end

Algorithm 63: Copiar