Algoritmos y Estructura de Datos II

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

Trabajo Práctico Diseño

Grupo 1

Integrante	LU	Correo electrónico
Bálsamo, Facundo	874/10	facundobalsamo@gmail.com
Lasso, Nicolás	763/10	lasso.nico@gmail.com
Rodriguez, Agustín	120/10	agustinrodriguez90@hotmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Módulo ArbolCategorias

1.1. Interfaz

parámetros formales

géneros acat
se explica con: ArbolDeCategorias

Operaciones

DAMECAT(**in** *dc*: `datosCat`)→ *res*: **Categoria**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_1(dc)$}
Complejidad: O(1)
Aliasing: La categoria se devuelve por referencia.

DAMEID(**in** *dc*: `datosCat`)→ *res*: **nat**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_2(dc)$}
Complejidad: O(1)
Aliasing: No tiene.

DAMEALTURA(**in** *dc*: `datosCat`)→ *res*: **nat**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_3(dc)$}
Complejidad: O(1)
Aliasing: No tiene.

DAMEHIJOS(**in** *dc*: `datosCat`)→ *res*: **itConj(puntero(datosCat))**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $crearIt(\pi_4(dc))$}
Complejidad: O(1)
Aliasing: No tiene?? ver post

DAMEPADRE(**in** *dc*: `datosCat`)→ *res*: **puntero(datosCat)**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_5(dc)$}
Complejidad: O(1)
Aliasing: Ver Alias

OBTENERAC(**in** *ac*: `acat`, **in** *c*: `Categoria`)→ *res*: **puntero(datosCat)**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $obtener(c, ac.familia)$}
Complejidad: O(|c|)
Aliasing: Ver Alias.

CATEGORIASAC(**in** ac: `acat`)→ res: **itLista(datosCat)**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} categorias(ac)\}$
Complejidad: O(1)
Aliasing: Ver alias. Eta bien el it a datosCat?

RAIZAC(**in** ac: `acat`)→ res: **Categoria**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} raiz(ac)\}$
Complejidad: O(1)
Aliasing: El nombre de la categoría raiz se pasa por referencia.

IDAC(**in** ac: `acat`, **in** c: `Categoria`)→ res: **nat**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} id(ac, c)\}$
Complejidad: O(|c|)
Aliasing: No tiene.

ALTURACATAC(**in** ac: `acat`, **in** c: `Categoria`)→ res: **nat**
Pre ≡ $\{esta?(c, ac)\}$
Post ≡ $\{res =_{\text{obs}} alturaCategoria(ac, c)\}$
Complejidad: O(|c|)
Aliasing: No tiene.

HIJOSAC(**in** ac: `acat`, **in** c: `Categoria`)→ res: **itConj(puntero(datosCat))**
Pre ≡ $\{esta?(c, ac)\}$
Post ≡ $\{res =_{\text{obs}} hijos(ac, c)\}$
Complejidad: O(|c|)
Aliasing: ver alias

PADREAC(**in** ac: `acat`, **in** c: `Categoria`)→ res: **Categoria**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} padre(ac, c)\}$
Complejidad: O(|c|)
Aliasing: El nombre de la categoría padre se pasa por referencia.

ALTURAAC(**in** ac: `acat`)→ res: **nat**
Pre ≡ $\{true\}$
Post ≡ $\{res =_{\text{obs}} altura(ac)\}$
Complejidad: O(1)
Aliasing: No tiene.

NUEVOAC(**in** c: `Categoria`)→ res: **acat**
Pre ≡ $\{\neg vacia?(c)\}$
Post ≡ $\{res =_{\text{obs}} nuevo(c)\}$
Complejidad: O(|c|)
Aliasing: Ver Alias

AGREGARAC(**in/out** ac: `acat`, **in** c: `categoria`, **in** h: `categoria`)
Pre $\equiv \{esta?(c, ac) \wedge \neg esta?(h, ac) \wedge \neg vacia?(h) \wedge ac_0 =_{\text{obs}} ac\}$
Post $\equiv \{ac =_{\text{obs}} agregar(ac_0, c, h)\}$
Complejidad: O(|c|+|h|)
Aliasing: No hay alias ya que no devuelve nada.

ESTA?(**in** c: `categoria`, **in** ac: `acat`)$\rightarrow$ res: **bool**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} esta?(c, ac)\}$
Complejidad: O(|c|)
Aliasing: No tiene.

ESSUBCATEGORIA(**in** ac: `acat`, **in** c: `categoria`, **in** h: `categoria`)$\rightarrow$ res: **bool**
Pre $\equiv \{esta?(c, ac) \wedge esta?(h, ac)\}$
Post $\equiv \{res =_{\text{obs}} esSubCategoria(ac, c, h)\}$
Complejidad: O(ver Complejidad y revisar parametros con especificacion)
Aliasing: No tiene.

fin interfaz

1.2. Representación

ArbolCategorias **se representa con** estrAC, donde estrAC es tupla<
raiz: `puntero(datosCat)`,
cantidad: `nat`,
alturaMax: `nat`,
familia: `diccTrie(Categoria, puntero(datosCat))`,
categorias: `Lista(datosCat)`>

datosCat es tupla<
categoria: `Categoria`,
id: `nat`,
altura: `nat`,
hijos: `Conj(puntero(datosCat))`,
padre: `puntero(datosCat)`>

1.2.1. Invariante de Representación

1.2.1.1. El Invariante Informalmente

1. Para cada '*padre* obtener el significado devolvera un puntero(datosCat) donde '*categoria*' es igual a la clave
2. Para toda clave '*padre*' que exista en '*familia*' debera ser o raiz o pertenecer a algun conjunto de punteros de '*hijos*' de alguna clave '*padre*'
3. Todos los elementos de '*hijos* de una clave '*padre*', cada uno de estos hijos tendran como '*abuelo*' a ese '*padre*' cuando sean clave.
4. '*cantidad*' sera igual a la longitud de la lista '*categorias*'.

5. Cuando la clave es igual a '*raiz*' la '*altura* es 1.
6. La '*altura*' del puntero a datosCat de cada clave es menor o igual a '*alturaMax*'.
7. Existe una clave en la cual, la '*altura*' del significado de esta es igual a '*alturaMax*'.
8. Los '*hijos*' de una clave tienen '*altura*' igual a 1 + '*altura* de la clave.
9. Todos los '*id*' de significado de cada clave deberan ser menor o igual a '*cant*'.
10. No hay '*id*' repetidos en el '*familia*.
11. Todos los '*id*' son consecutivos.

1.2.1.2. El Invariante Formalmente

Rep : estrAC $\rightarrow$ boolean

($\forall\, ac$: `estrAC`) Rep(ac) $\equiv$ true $\Longleftrightarrow$

1. $(\forall\, x$: `string`$)(def?(x, e.familia)) \Leftrightarrow (*obtener(x, e.familia)).categoria = x \wedge_{\mathrm{L}}$
2. $(\forall\, x, y$: `string`$)(def?(x, e.familia)) \Leftrightarrow (x == e.raiz) \vee (def?(y, e.familia)) \wedge_{\mathrm{L}}$
   $x \in hijosDe(*((obtener(y, e.familia))).hijos) \wedge_{\mathrm{L}}$
3. $(\forall\, x, y$: `string`$)(def?(x, e.familia)) \wedge (def?(y, e.familia)) \Rightarrow_{L}$
   $y \in *((obtener(x, e.familia))).hijos \Leftrightarrow$
   $(*(*(obtener(y, e.familia))).padre).categoria = x \wedge_{\mathrm{L}}$
4. $e.cantidad = longitud(e.categorias) \wedge_{\mathrm{L}}$
5. $(\forall\, x$: `string`$)(def?(x, e.familia)) \wedge x = e.raiz \Rightarrow_{L}$
   $(*(obtener(x, e.familia))).altura = 1 \wedge_{\mathrm{L}}$
6. $(\forall\, x$: `string`$)(def?(x, e.familia)) \Rightarrow_{L} (*obtener(x, e.familia)).altura \leq e.alturaMax \wedge_{\mathrm{L}}$
7. $(\exists\, x$: `string`$)(def?(x, e.familia)) \wedge_{\mathrm{L}} *((obtener(x, e.familia))).altura = e.alturaMax \wedge_{\mathrm{L}}$
8. $(\forall\, x, y$: `string`$)(def?(x, e.familia)) \wedge (def?(y, e.familia)) \wedge_{\mathrm{L}}$
   $y \in hijosDe((*(obtener(x, e.familia))).hijos) \Rightarrow$
   $(*(obtener(y, e.familia))).altura = 1 + (*(obtener(x, e.familia))).altura \wedge_{\mathrm{L}}$
9. $(\forall\, x$: `string`$)(def?(x, e.familia)) \Rightarrow_{L} (*(obtener(x, e.familia))).id \leq e.cant \wedge_{\mathrm{L}}$
10. $(\forall$**x,y** $string$: `)`$(def?(x, e.familia)) \wedge (def?(y, e.familia)) \Rightarrow_{L}$
    $(*(obtener(x, e.familia))).id \neq (*(obtener(y, e.familia))).id \wedge_{\mathrm{L}}$
11. $(\forall$**x** $string$: `)`$(def?(x, e.familia))(\exists\, y$: `string`$)(def?(y, e.familia)) \Leftrightarrow$
    $(*(obtener(y, e.familia))).id \leq e.cantidad \wedge (*(obtener(x, e.familia))).id < e.cantidad \wedge_{\mathrm{L}}$
    $(*(obtener(y, e.familia))).id = 1 + (*(obtener(x, e.familia))).id$

1.2.2. Función de Abstracción

Abs : e: `estrAC` $\rightarrow$ acat Rep(e)

($\forall e$: `estrAC`) Abs(e) $=_{\text{obs}}$ ac: acat |

1. $categorias(ac) =_{\text{obs}} todasLasCategorias(e.categorias) \wedge_{\text{L}}$
2. $raiz(ac) =_{\text{obs}} (*e.raiz).categoria \wedge_{\text{L}}$
3. $(\forall c$: `Categoria`$)esta?(c, ac) \wedge c \neq raiz(ac) \Rightarrow_L$
   $padre(ac, c) = (*(*(obtener(c, e.familia))).padre).categoria \wedge_{\text{L}}$
4. $(\forall c$: `Categoria`$)esta?(c, ac) \Rightarrow_L id(ac, c) = (*(obtener(c, e.familia))).id$

1.2.2.1. Funciones auxiliares

todasLasCategorias : secu(datosCat) $\longrightarrow$ conj(categoria)
todasLasCategorias $\equiv$ Ag((prim(cs)).categoria,fin(cs))

1.3. Algoritmos

Algoritmo 1 iDameCat

```
1: function IDAMECAT(in dc: datosCat)→ res: Categoria
2:     res ← dc.categoria                                   //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 2 iDameId

```
1: function IDAMEID(in dc: datosCat)→ res: nat
2:     res ← dc.id                                          //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 3 iDameAltura

```
1: function IDAMEALTURA(in dc: datosCat)→ res: Categoria
2:     res ← dc.altura                                      //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 4 iDameHijos

```
1: function IDAMEHIJOS(in dc: datosCat)→ res: itConj(puntero(datosCat))
2:     res ← crearIt(dc.hijos)                              //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 5 iDamePadre

```
1: function IDAMEPADRE(in dc: datosCat)→ res: puntero(datosCat)
2:     res ← dc.padre                                          //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 6 iObtenerAC

```
1: function IOBTENERAC(in ac: estrAC, in c: Categoria)→ res: puntero(datosCat)
2:     res ← obtener(c,ac.familia)                             //O(|c|)
3: end function
```

Complejidad: O(|c|)

Algoritmo 7 iCategoriasAC

```
1: function ICATEGORIASAC(in ac: estrAC)→ res: itLista(datosCat)
2:     res ← crearIt(ac.categorias)                            //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 8 iRaizAC

```
1: function IRAIZ(in ac: estrAC)→ res: Categoria
2:     res ← (*(ac.raiz)).categoria                            //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 9 iIdAC

```
1: function ID(in ac: estrAC, in c: Categoria)→ res: nat
2:     res ← ((*obtener(c,ac.familia)).id                      //O(|c|)
3: end function
```

Complejidad: O(|c|)

Algoritmo 10 iAlturaCatAC

```
1: function IALTURACATAC(in ac: estrAC, in c: Categoria)→ res: nat
2:     res ← (*obtener(c,ac.familia)).altura                   //O(|c|)
3: end function
```

Complejidad: O(|c|)

Algoritmo 11 iHijosAC

```
1: function IHIJOSAC(in ac: estrAC, in c: Categoria)→ res: itConj(puntero(datosCat))
2:     res ← crearIt((*obtener(c,ac.familia)).hijos)           //O(|c|)
3: end function
```

Complejidad: O(|c|)

Algoritmo 12 iPadreAC

```
1: function iPadreAC(in ac: estrAC, in c: Categoria)→ res: Categoria
2:     res ← (*(*obtener(c,ac.familia)).abuelo).categoria                    //O(|c|)
3: end function
```

Complejidad: O(|c|)

Algoritmo 13 iAlturaAC

```
1: function iAlturaAC(in ac: estrAC)→ res: nat
2:     res ← ac.alturaMax                                                    //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 14 iNuevoAC

```
1: function iNuevoAC(in c: Categoria)→ res: estrAC
2:     res.cantidad ← 1                                                      //O(1)
3:     datosCat tuplaA                                                       //O(1)
4:     tuplaA ← tupla(c,1,1,vacio(),Null)                                    //O(|c|)
5:     puntero(datosCat) punt ← &tuplaA                                      //O(1)
6:     res.raiz ← punt                                                       //O(1)
7:     res.alturaMax ← 1                                                     //O(1)
8:     definir(c, punt, res.familia)                                         //O(|c|)
9:     agregarAtras(tuplaA, res.categorias)                                  //O(1)
10: end function
```

Complejidad: O(|c|)

Algoritmo 15 iAgregarAC

```
1: function iAgregarAC(in/out ac: estrAC, in c: Categoria, in h: Categoria)
2:     puntero(datosCat) puntPadre ← obtener(c,ac.familia)                   //O(|c|)
3:     if (*puntPadre).altura == ac.alturaMax then                           //O(1)
4:         ac.alturaMax++                                                    //O(1)
5:     end if
6:     datosCat tuplaA ← (h,ac.cantidad+1,(*puntPadre).altura+1,vacio(),puntPadre)   //O(|h|)
7:     puntero(datosCat) punt ← &tuplaA                                      //O(1)
8:     Agregar((*puntPadre).hijos,punt)                                      //O(1)
9:     definir(h,punt,ac.familia)                                            //O(|h|)
10:    ac.cantidad++                                                         //O(1)
11:    agregarAtras(tuplaA,ac.categorias)                                    //O(1)
12: end function
```

Complejidad: O(|c|+|h|)

Algoritmo 16 iEsta?

```
1: function iEsta?(in ac: estrAC, in c: Categoria)→ res: bool
2:     res ← def?(c,ac.familia)                                              //O(|c|)
3: end function
```

Complejidad: O(|c|)

Algoritmo 17 iEsSubCategoria

```
1: function iEsSubCategoria(in ac: estrAC, in c: Categoria, in h: Categoria)→ res: bool
2:     res ← false                                                              //O(1)
3:     if h == c then                                                           //O(|h|)
4:         res ← true                                                           //O(1)
5:     else
6:         if h == raizAC(ac) then                                              //O(|h|)
7:             res ← false                                                      //O(1)
8:         else
9:             puntero(datosCat) actual ← (*obtener(h,ac.familia)).padre        //O(|h|)
10:            puntero(datosCat) puntC ← (*obtener(c,ac.familia)).padre         //O(|c|)
11:            while res == false ∧ actual ≠ NULL do                            //O(alturaAC(ac))
12:                if puntC == actual then                                      //O(|c|)
13:                    res ← true                                               //O(1)
14:                else
15:                    actual ← (*actual).padre                                 //O(1)
16:                end if
17:            end while
18:        end if
19:    end if
20: end function
```

Complejidad: O(|h| + |c| + alturaAC(ac))

1.4. Analisis de complejidades

1. **iAutor**

   Se pasa una referencia al autor del mensaje en O(1)

   Orden Total: O(1)

2. **iContenido**

   Se pasa una referencia al contenido del mensaje en O(1)

   Orden Total: O(1)

3. **iCreado**

   Se copia la fecha del mensaje en O(1)

   Orden Total: O(1)

4. **iTags**

   Se pasa una referencia a la lista de tags del mensaje en O(1)

   Orden Total: O(1)

2. Módulo LinkLinkIt

2.1. Interfaz

parámetros formales

géneros linkLinkIt
se explica con: TAD linkLinkIt

Operaciones

DAMELINK(**in** *dl*: `datosLink`)→ *res*: **Link**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_1(dl)$}
Complejidad: O(1)
Aliasing: El link se devuelve por referencia.

DAMECATDLINK(**in** *dl*: `datosLink`)→ *res*: **puntero(datosCat)**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_2(dl)$}
Complejidad: O(1)
Aliasing: Ver Alis

DAMEACCESOS(**in** *dl*: `datosLink`)→ *res*: **itLista(acceso)**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $crearIt(\pi_3(dl))$}
Complejidad: O(1)
Aliasing: Ver Alias

DAMECANTACCESOS(**in** *dl*: `datosLink`)→ *res*: **nat**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_4(dl)$}
Complejidad: O(1)
Aliasing: No tiene

DAMEDIA(**in** *a*: `acceso`)→ *res*: **Fecha**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_1(a)$}
Complejidad: O(1)
Aliasing: No tiene

DAMECANTA(**in** *a*: `acceso`)→ *res*: **nat**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ $\pi_2(a)$}
Complejidad: O(1)
Aliasing: No tiene

DAMEACATLLI(**in** *lli*: `linkLinkIt`)→ *res*: **acat**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ *lli.arbolCategorias*}
Complejidad: O(1)
Aliasing: res es una referencia a lli.arbolCategorias

CATEGORIASLLI(**in** *lli*: `linkLinkIt`)→ *res*: **itLista(datosCat)**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ *categorias*(*lli.arbolCategorias*)}
Complejidad: O(1)
Aliasing: Ver Alias

FECHAACTUAL(**in** *lli*: `linkLinkIt`)→ *res*: **Fecha**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ *fechaActual*(*lli*)}
Complejidad: O(1)
Aliasing: No tiene

LINKSLLI(**in** *lli*: `linkLinkIt`)→ *res*: **itLista(datosLink)**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ *links*(*lli*)}
Complejidad: O(1)
Aliasing: Ver Alias y ver post, como en categorias(acat)

CATEGORIALINK(**in** *lli*: `linkLinkIt`, **in** *l*: `Link`)→ *res*: **Categoria**
Pre ≡ {$l \in links(lli)$}
Post ≡ {*res* =$_{\text{obs}}$ *categoriaLink*(*lli*, *l*)}
Complejidad: O(|l|)
Aliasing: La categoria se devuelve por referencia.

INICIARLLI(**in** *ac*: `acat`)→ *res*: **linkLinkIt**
Pre ≡ {*true*}
Post ≡ {*res* =$_{\text{obs}}$ *iniciar*(*ac*)}
Complejidad: O(#categorias(ac))
Aliasing: VerAlias

NUEVOLINKLLI(**in/out** *lli*: `linkLinkIt`, **in** *l*: `Link`, **in** *c*: `Categoria`)
Pre ≡ {$c \in categorias(lli) \land l \notin links(lli) \land \neg vacia?(l) \land lli_0 = lli$}
Post ≡ {$lli = nuevoLink(lli_0, l, c)$}
Complejidad: O(|l|+|c|+altura(lli.arbolCategorias))
Aliasing: No hay alias ya que no devuelve nada.

ACCEDERLLI(**in/out** *lli*: `linkLinkIt`, **in** *l*: `Link`, **in** *f*: `Fecha`)
Pre ≡ {$l \in links(lli) \land f \geq fechaActual(lli) \land lli_0 = lli$}
Post ≡ {$lli = acceso(lli_0, l, f)$}
Complejidad: O(|l|)
Aliasing: No hay alias ya que no devuelve nada.

esReciente?(**in** *lli*: `linkLinkIt`, **in** *l*: `Link`, **in** *f*: `Fecha`)→ *res*: **bool**
Pre ≡ {$l \in links(lli)$}
Post ≡ {$res =_{\text{obs}} esReciente?(s, l, f)$}
Complejidad: O(|l|)
Aliasing: VerAlias

accesosRecientes(**in** *lli*: `linkLinkIt`, **in** *c*: `Categoria`, **in** *l*: `Link`)→ *res*: **nat**
Pre ≡ {$c \in categorias(lli) \land l \in links(lli)$}
Post ≡ {$res =_{\text{obs}} accesosRecientes(lli, c, l)$}
Complejidad: O(|c|+|l| ??)
Descripción: No tiene.

linksOrdenadosPorAccesos(**in** *lli*: `linkLinkIt`, **in** *c*: `Categoria`)→ *res*: **Secu(Link)**
Pre ≡ {$c \in categorias(lli)$}
Post ≡ {$res =_{\text{obs}} linksOrdenadosPorAccesos(lli, c)$}
Complejidad: O(n^2 ???(Que es n? corregir))
Aliasing: VerAlias

cantLinks(**in** *lli*: `linkLinkIt`, **in** *c*: `Categoria`)→ *res*: **nat**
Pre ≡ {$c \in categorias(lli)$}
Post ≡ {$res =_{\text{obs}} cantLinks(lli, c)$}
Complejidad: O(|c|)
Aliasing: No tiene.

menorReciente(**in** *lli*: `linkLinkIt`, **in** *l*: `Link`)→ *res*: **Fecha**
Pre ≡ {$l \in links(lli)$}
Post ≡ {$res =_{\text{obs}} menorReciente(lli, l)$}
Complejidad: O(|l|)
Aliasing: No tiene.

diasRecientes(**in** *lli*: `linkLinkIt`, **in** *l*: `Link`)→ *res*: **Fecha**
Pre ≡ {$l \in links(lli)$}
Post ≡ {$res =_{\text{obs}} diasRecientes(lli, l)$}
Complejidad: O(|l|)
Aliasing: No tiene.

diasRecientesDesde(**in** *lli*: `linkLinkIt`, **in** *l*: `Link`)→ *res*: **Fecha**
Pre ≡ {$l \in links(lli)$}
Post ≡ {$res =_{\text{obs}} diasRecientesDesde(lli, l)$}
Complejidad: O(|l|)
Aliasing: No tiene.

diasRecientesParaCategorias(**in** *lli*: `linkLinkIt`, **in** *c*: `Categorias`)→ *res*: **Conj(Fecha)**
Pre ≡ {$c \in categorias(lli)$}
Post ≡ {$res =_{\text{obs}} diasRecientesParaCategorias(lli, c)$}
Complejidad: O(|c|)
Aliasing: VerAlias

LINKCONULTIMOACCESO(**in** lli: `linkLinkIt`, **in** c: `Categoria`, **in** lc: `Conj(Link)`)$\rightarrow$ res: **Link**
Pre $\equiv \{c \in categorias(lli)\}$
Post $\equiv \{res =_{\text{obs}} linkConUltimoAcceso(lli, c, lc)\}$
Complejidad: O(Complejidad?)
Aliasing: VerAlias

SUMARACCESOSRECIENTES(**in** lli: `linkLinkIt`, **in** l: `Link`, **in** fc: `Conj(Fecha)`)$\rightarrow$ res: **nat**
Pre $\equiv \{l \in links(lli) \land fs \subseteq diasRecientes(lli, l)\}$
Post $\equiv \{res =_{\text{obs}} sumarAccesosRecientes(lli, l, fs)\}$
Complejidad: O(|l|)
Aliasing: No tiene.

BUSCARMAX(**in** ls: `Lista(puntero(datosLink))`)$\rightarrow$ res: **itLista(puntero(datosLink))**
Pre $\equiv \{ls =_{\text{obs}} ls_0\}$
Post $\equiv \{res =_{\text{obs}} buscarMax(ls_0)\}$
Complejidad: O(n????)
Aliasing: Alias?????

ESTAORDENADA(**in** ls: `Lista(puntero(datosLink))`)$\rightarrow$ res: **bool**
Pre $\equiv \{ls =_{\text{obs}} ls_0\}$
Post $\equiv \{res =_{\text{obs}} estaOrdenada(ls_0)\}$
Complejidad: O(n????)
Aliasing: Alias?????

FECHAULTIMOACCESO(**in** lli: `linkLinkIt`, **in** l: `Link`)$\rightarrow$ res: **Fecha**
Pre $\equiv \{l \in links(lli)\}$
Post $\equiv \{res =_{\text{obs}} fechaUltimoAcceso(lli, l)\}$
Complejidad: O(|l|)
Aliasing: No tiene

ACCESOSRECIENTESDIA(**in** lli: `linkLinkIt`, **in** l: `Link`, **in** f: `Fecha`)$\rightarrow$ res: **nat**
Pre $\equiv \{l \in links(lli)\}$
Post $\equiv \{res =_{\text{obs}} accesosRecientesDia(lli, l, f)\}$
Complejidad: O(|l|)
Aliasing: No tiene

fin interfaz

2.2. Representación

LinkLinkIt **se representa con** estrLLI, donde estrLLI es tupla<
arbolCategorias: `acat`,
actual: `Fecha`,
linkInfo: `diccTrie(Link,puntero(datosLink))`,
listaLink: `Lista(datosLink)`,
arrayCatLinks: `arreglo-dimen()linksFamilia`>

datosLink es tupla<
link: `Link`,
catDLink: `puntero(datosCat)`,
accesosRecientes: `Lista(acceso)`,
cantAaccesosRecientes: `nat`>

acceso es tupla<
dia: `Fecha`,
cantAccesos: `nat`>

linksFamilia es Lista(puntero(datosLink))

Las Reglas se representan con los países como posiciones en el vector, ignorando la posición "0". En la posición que representa un país, se tiene una lista de países que este no puede ver. Se utiliza el módulo vector dado su costo amortizado de inserción de nuevos países, el cual corresponde con la complejidad pedida.

También contamos con 2 estructuras auxiliares:

MatrizRestricciones es una matriz cuadrada que tiene la relacion de todos los paises. MatrizRelaciones[a][b] indica si el pais .ªrestringe al "b". Esto nos permite consultar por cualquier restricción en O(1)

ArregloRelaciones es un arreglo con todos los países y contiene la lista de paises que puede ver y de lo que los pueden ver. Esto permite comprobar si existe algún hueco que incluye a un pais "p.^{en} O(#puedenVerA(p) + #puedeVer(p)).

Estas estructuras son creadas únicamente cuando se llama a la función tieneAlgúnHueco? y luego descartadas.

2.2.1. Invariante de Representación

2.2.1.1. El Invariante Informalmente

1. Para todo '*link*' que exista en '*accesosXLink*' la '*catDLink*' de la tupla apuntada en el significado debera existir en *'arbolCategorias'*.

2. Para todo '*link*' que exista en '*accesosXLink*', todos los *dia*' de la lista '*accesosRecientes*' deberan ser menor o igual a *actual*, estan ordenados,no hay dias repetidos y la longitud de la lista es menor o igual a 3.

3. Para todo '*link*' que exista en '*accesosXLink*' su significado deberá existir en '*listaLinks*' y viceversa.

4. Para todo '*link*' que exista en '*accesosXLink*' su significado deberá aparecer en '*arrayCantLinks*' en la posicion igual al id de '*catDLink*' y en las posiciones de los predecesores de esa categoria y en ninguna otra.

5. No hay 2 claves que existan en '*accesosXLink*' y devuelvan el mismo significado.
6. No existen '*link*' repetidos en las tuplas de '*listaLinks*'.
7. No hay elementos repetidos en ninguna lista '*linksFamilia*'.
8. Para todo '*link*' que exista en '*accesosXLink*', '*cantAccesosRecientes*' es igual a la suma de '*cantAccesos*' de cada elemento de la lista '*accesosRecientes*'

2.2.1.2. El Invariante Formalmente

Rep : estrLLI $\rightarrow$ boolean

$(\forall\, lli$: `estrLLI`) Rep(lli) $\equiv$ true $\Longleftrightarrow$

1. $(\forall\, l$: `Link`$)(def?(l, lli.accesosXLink)) \Rightarrow_L$
   $(*obtener(l, lli.accesosXLink)).catDLink \in todasLasCategorias(lli.arbolCategorias.categorias)$
   $\wedge_{\text{L}}$
2. $(\forall\, l$: `Link`$)(def?(l, lli.accesosXLink)) \Rightarrow_L$
   $long((*obtener(l, lli.accesosXLink)).accesosRecientes) \leq 3 \wedge$
   $accesoOrdenadoNoRepetido((*obtener(l, lli.accesosXLink)).accesosRecientes) \wedge_{\text{L}}$
   $fechasCorrectas(lli.actual, ((*obtener(l, lli.accesosXLink)).accesosRecientes)) \wedge_{\text{L}}$
3. $Faltaelrep3 - 4aca \wedge_{\text{L}}$
4. $(\forall\, l$: `Link`$)(def?(l, lli.accesosXLink)) \Rightarrow_L$
   $((\forall\, c$: `Categoria`$)c \in todasLasCategorias(lli.arbolCategorias.categorias) \Rightarrow_L$
   $(esta?((obtener(l, lli.accesosXLink)), arrayCatLinks[id(c, lli.arbolCategorias)]) \Leftrightarrow$
   $esPredecesor(c, (*obtener(l, lli.accesosXLink)).categoria, lli.arbolCategorias))) \wedge_{\text{L}}$
5. $(\forall\, l, l'$: `Link`$)l \neq l' \wedge (def?(l, lli.accesosXLink)) \wedge (def?(l', lli.accesosXLink)) \Rightarrow_L$
   $(*obtener(l, lli.accesosXLink)) \neq (*obtener(l', lli.accesosXLink)) \wedge_{\text{L}}$
6. $(\forall$**i,i'** nat: `)`$i < long(lli.listaLinks) \wedge i' < long(lli.listaLinks) \Rightarrow_L$
   $lli.listaLinks_i.link = lli.listaLinks_{i'}.link \Leftrightarrow i = i' \wedge_{\text{L}}$
7. $(\forall\, i$: `nat`$)i < tam(lli.arrayCatLinks) \Rightarrow_L sinRepetidos(linksFamilia_i) \wedge_{\text{L}}$
8. $(\forall\, l$: `Link`$)(def?(l, lli.accesosXLink)) \Rightarrow_L$
   $(*obtener(l, lli.accesosXLink)).cantAccesosRecientes ==$
   $cantidadDeAccesos((*obtener(l, lli.accesosXLink)).accesosRecientes)$

2.2.2. Función de Abstracción

Abs : e: `estrLLI` → linkLinkIt Rep(e)

($\forall e$: `estrLLI`) Abs(e) $=_{\text{obs}}$ lli: linkLinkIt |

1. $categorias(lli) = categorias(e.arbolCategorias) \wedge$
2. $links(lli) = todosLosLinks(e.listaLinks) \wedge_{\text{L}}$
3. $(\forall l$: `Link`$)def?(l, e.linkInfo) \Rightarrow_{L}$
   $categoriaLink(lli, l) = (*obtener(l, e.linkInfo))catDLink \wedge$
4. $fechaActual(lli) = e.actual \wedge$
5. $(\forall l$: `Link`$)l \in links(e) \Rightarrow_{L}$
   $fechaUltimoAcceso(lli, l) = (ultimo((*obtener(l, e.linkInfo)).accesosRecientes)).dia$
   $\wedge$
6. $(\forall l$: `Link`$)(\forall f$: `Fecha`$)l \in links(lli) \wedge_{\text{L}} esReciente?(e, l, f) \Rightarrow_{L}$
   $accesosRecientesDia(lli, l, f) =$
   $cantidadPorDia(f, (*obtener(l, e.linkInfo)).accesosRecientes)$

2.2.2.1. Funciones auxiliares

2.3. Algoritmos

Algoritmo 18 iDameLink

```
1: function IDAMELINK(in dl: datosLink)→ res: Link
2:     res ← dl.link                                    //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 19 iDameCatDLink

```
1: function IDAMECATDLINK(in dl: datosLink)→ res: puntero(datosCat)
2:     res ← dl.catDLink                                //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 20 iDameAccesos

```
1: function IDAMEACCESOS(in dl: datosLink)→ res: itLista(acceso)
2:     res ← crearIT(dl.accesosRecientes)               //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 21 iDameCantAccesos

```
1: function IDAMECANTACCESOS(in dl: datosLink)→ res: nat
2:     res ← dl.cantAccesosRecientes                    //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 22 iDameDia

```
1: function iDameDia(in a: acceso)→ res: Fecha
2:     res ← a.dia                                                    //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 23 iDameCantA

```
1: function iDameCantA(in a: acceso)→ res: nat
2:     res ← a.cantAccesos                                            //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 24 idameACatLLI

```
1: function idameACatLLI(in lli: estrLLI)→ res: acat
2:     res ← lli.arbolCategorias                                      //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 25 icategoriasLLI

```
1: function icategoriasLLI(in lli: estrLLI)→ res: acat
2:     res ← categorias(lli.arbolCategorias)                          //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 26 iFechaActual

```
1: function ifechaActual(in lli: estrLLI)→ res: Fecha
2:     res ← lli.actual                                               //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 27 iLinksLLI

```
1: function ilinksLLI(in lli: estrLLI)→ res: itLista(datosLink)
2:     res ← crearIt(lli.listaLinks)                                  //O(1)
3: end function
```

Complejidad: O(1)

Algoritmo 28 iCategoriaLink

```
1: function icategoriaLink(in lli: estrLLI, in l: Link)→ res: Categoria
2:     res ← (*obtener(l,lli.linksInfo)).catDLink                     //O(|l|)
3: end function
```

Complejidad: O(|l|)

Algoritmo 29 iIniciarLLI

```
1:  function IINICIARLLI(in ac: acat)→ res: estrLLI
2:      res.actual ← 1                                                    //O(1)
3:      res.arbolCategorias ← &ac                                         //O(1)
4:      nat c ← 1                                                         //O(1)
5:      res.arrayCatLinks ← crearArreglo(#categoriasAC(ac))               //O(#categoriasAC(ac))
6:      res.listaLinks ← vacia()                                          //O(1)
7:      res.linksInfo ← vacio()                                           //O(1)
8:      while c ≤ #categoriasAC(ac) do                                    //O(#categoriasAC(ac))
9:          linksFamilia llist ← vacia()                                  //O(1)
10:         res.arrayCatLinks[c] ← llist                                  //O(1)
11:         c++                                                           //O(1)
12:     end while
13: end function
```

Complejidad: O(#categoriasAC(ac))

Algoritmo 30 iNuevoLink

```
1:  function INUEVOLINK(in/out lli: estrLLI, in l: Link, in c: Categoria)
2:      puntero(datosCat) cat ← obtener(c,lli.arbolCategorias)            //O(|c|)
3:      Lista(acceso) accesoDeNuevoLink ← vacia()                         //O(1)
4:      datosLink nuevoLink ← <l,cat,accesoDeNuevoLink,0>                 //O(|l|)
5:      puntero(datosLink) puntLink ← nuevoLink                           //O(1)
6:      definir(l,puntLink,lli.linkInfo)                                  //O(|l|)
7:      agregarAtras(lli.listaLinks,nuevoLink)                            //O(1)
8:      while cat ≠ NULL do                                               //O(alturaAC(ac))
9:          agregarAtras(lli.arrayCatLinks[dameId(cat)],puntLink)         //O(1)
10:         cat ← damePadre(cat)                                          //O(1)
11:     end while
12: end function
```

Complejidad: O(|c|+|l|+alturaAC(ac))

Algoritmo 31 iAccederLLI

```
1:  function IACCEDERLLI(in/out lli: estrLLI, in l: Link, in f: Fecha)
2:      if lli.actual ≠ f then                                            //O(1)
3:          lli.actual ← f                                                //O(1)
4:      end if
5:      puntero(datosLink) puntLink ← obtener(l,lli.linkInfo)             //O(|l|)
6:      if ultimo((*puntLink).accesos).dia == f then                      //O(1)
7:          ultimo((*puntLink).accesos).cantAccesos ++                    //O(1)
8:      else
9:          agregarAtras((*puntLink).accesos,<1,f>)                       //O(1)
10:     end if
11:     if longitud((*puntLink).accesos) == 4 then                        //O(1)
12:         fin((*puntLink).accesos)                                      //O(1)
13:     end if
14:     (*puntLink).cantAccesosRecientes++                                //O(1)
15: end function
```

Complejidad: O(|l|)

Algoritmo 32 iFechaUltimoAcceso

```
1: function IFECHAULTIMOACCESO(in lli: estrLLI, in l: Link)→ res: Fecha
2:     res ← (ultimo((*obtener(l,lli.linkInfo)).accesosRecientes)).dia          //O(|l|)
3: end function
```

Complejidad: O(|l|)

Algoritmo 33 iAccesosRecientesDia

```
1: function IACCESOSRECIENTESDIA(in lli: estrLLI)→ res: nat
2:     Lista(acceso) accesos ← vacia()                                          //O(1)
3:     res ← 0                                                                  //O(1)
4:     accesos ← (*obtener(l,lli.linkInfo)).accesosRecientes                    //O(|l|)
5:     while esVacia?(accesos) ∧ res=0 do                //O(|accesos|) = O(1)
6:         if ultimo(accesos).dia == f then
7:             res ← ultimo(accesos).cantAccesos                                //O(1)
8:         else
9:             accesos ← fin(accesos)                                           //O(1)
10:        end if
11:    end while
12: end function
```

Complejidad: O(|l|)

Algoritmo 34 iEsReciente

```
1: function IESRECIENTE(in lli: estrLLI, in l: Link, in f: Fecha)→ res: bool
2:     res ← menorReciente(s,l) ≤ f ∧ f ≤ fechaUltimoAcceso(s,l)                //O(|l)
3: end function
```

Complejidad: O(|l|)

Algoritmo 35 iAccesosRecientes

```
1: function IACCESOSRECIENTES(in lli: estrLLI, in c: Categoria, in l: Link)→ res: nat
2:     hacer
3: end function
```

Complejidad: O(ver)

Algoritmo 36 iLinksOrdenadosPorAccesos

```
1: function ILINKSORDENADOSPORACCESOS(in lli : estrLLI, in c : Categoria)→ res : itListaUni(Lista(Link))
2:     nat id ← id(lli.arbolCategorias,c)                                          //O(|c|)
3:     Lista(puntero(datosLink)) listaOrdenada ← vacia()                           //O(1)
4:     itLista(puntero(datosLink)) itMax ← crearIt(lli.arrayCatLinks[id])          //O(1)
5:     if estaOrdenada?(lli.arrayCatLinks[id]) then          //O(longitud(lli.arrayCatLinks[id]))
6:         while haySiguiente?(lli.arrayCatLinks[id]) do     //O(longitud(lli.arrayCatLinks[id]))
7:             itMax ← buscarMax(lli.arrayCatLinks[id])      //O(longitud(lli.arrayCatLinks[id]))
8:             agregarAtras(listaOrdenada,siguiente(itMax))                        //O(1)
9:             eliminarSiguiente(itMax)                                            //O(1)
10:        end while
11:        lli.arrayCatLinks[id] ← listaOrdenada
12:    else
13:        res ← crearIt(lli.arrayCatLinks[id])
14:    end if
15: end function
```

Complejidad: O($(longitud(lli.arrayCatLinks[id]))^2$ + |c|)

Algoritmo 37 iBuscarMax

```
1: function IBUSCARMAX(in ls : Lista(puntero(datosLink)))→ res : itLista(puntero(datosLink))
2:     rehacer
3: end function
```

Complejidad: O(longitud(ls))

Algoritmo 38 iEstaOrdenada

```
1: function IESTAORDENADA(in ls: Lista(puntero(datosLink)))→ res: bool
2:     res ← true                                                                  //O(1)
3:     itLista(puntero(datosLink)) itRecorre ← crearIt(ls)                         //O(1)
4:     nat aux ← (*siguiente(itRecorre)).cantAccesosRecientes                      //O(1)
5:     while haySiguiente(itRecorre) ∧ res==true do                                //O(n)
6:         avanzar(itRecorre)                                                      //O(1)
7:         if aux <(*siguiente(itRecorre)).cantAccesosRecientes then               //O(1)
8:             res ← false                                                         //O(1)
9:         end if
10:        aux ← (*siguiente(itRecorre)).cantAccesosRecientes                      //O(1)
11:    end while
12: end function
```

Complejidad: O(n)

Algoritmo 39 iCantLinks

```
1: function ICANTLINKS(in lli: estrLLI, c: Categoria)→ res: nat
2:     puntero(datosCat) cat ← obtener(c,lli.arbolCategorias)                      //O(|c|)
3:     res ← longitud(lli.arrayCatLinks[(*cat).id])                                //O(1)
4: end function
```

Complejidad: O(|c|)

Algoritmo 40 iMenorReciente

```
1: function IMENORRECIENTE(in lli: estrLLI, l: Link)→ res: Fecha
2:     res ← max(fechaUltimoAcceso(lli,l)+1,diasRecientes)-diasRecientes        //O(|l|)
3: end function
```

Complejidad: O(|l|)

Algoritmo 41 iDiasRecientes

```
1: function IDIASRECIENTES(in lli: estrLLI, l: Link)→ res: Conj(Fecha)
2:     puntero(datosCat) cat ← diasRecientesDesde(lli,l,menorReciente(lli,l))   //O(|l|)
3: end function
```

Complejidad: O(|l|)

2.4. Analisis de complejidades

1. **iVacías**

   Se crea 2 vectores vacío y se agrega atras uno de otro. Esto cuesta O(1).

   Orden Total: O(1)

2. **iAgregarPaís**

   Se crea una Lista Vacía, lo cual tarda O(1), y se agrega esta lista al final del vector. Esta última operación cuesta O(1) amortizando los casos en los que el vector debe redimensionarse.

   Orden Total:O(1+1)amortizado = **O(1)amortizado**

3. **iRestringirPaís**

   Se obtiene una lista de una coordenada del vector, lo cual cuesta O(1). En dicha lista, se agrega un país al final, lo cual cuesta copiar el elemento, pero como el elemento de tipo País es un renombre de Nat, la operación cuesta O(1).

   Orden Total: O(1)+O(1)=**O(1)**

4. **iPaises**

   Se crea un conjunto vacío, lo cual tarda O(1). Se crean también dos variables Nat y se les asigna un valor, también en O(1) (obtener la longitud del vector tarda O(1)). Luego, se entra a un ciclo que itera una cantidad de veces igual a la cantidad de países. En este ciclo, se llama a AgregarRápido del conjunto, que tarda el costo de copiar el elemento, pero al ser un Nat, cuesta O(1). Luego se avanza el contador, lo cual cuesta O(1) por ser una operación con Nats. Al salir del ciclo, se devuelve el conjunto.

   Orden Total: O(1)+O(1)+ O(1+1)+O(CP*O(1+1)) = **O(CP)**

5. **iPuedeVer?**

   Se obtiene una lista de una coordenada del vector, lo cual cuesta O(1), y se crea un iterador a ella, también en O(1). Se asigna "true.ª res en O(1). Luego, se entra a un ciclo. Este ciclo itera hasta recorrer todos los países de la lista, o hasta encontrar el país buscado. En el peor caso, si están listados todos los países y el buscado es el último, el ciclo itera una cantidad de veces igual a la cantidad de países. en este ciclo, se llama a Siguiente del iterador, se hace una comparación de Nat, se asigna el valor de la comparación a res, y se avanza el iterador. Todas estas operaciones cuestan O(1).

   Orden Total: O(1+1)+O(1)+O(CP*(O(1+1+1)+O(1))) = **O(CP)**

6. **inuevaMatriz**

   Se crean dos variables nat en O(1). Se le asigna 0 a una de estas variables, también en O(1). Se crea un arreglo de tamaño igual a la cantidad de países en O(CP) y se asigna este arreglo a res en O(1). Luego, se entra a un ciclo que itera un número de veces igual a la cantidad de países. En cada iteración, se crea un arreglo de tamaño cantidad países en O (CP) y se define este nuevo arreglo en la coordenada del areglo res en O(1). Luego, se asigna 0 a la segunda variable nat en O(1). Se entra a un ciclo interno que itera un número de veces igual a la cantidad de países, y en cada iteración asigna "False.ª una coordenada del subarreglo en una coordenada del arreglo res en O(1). Luego se avanza el contador p2, y fuera de este ciclo, se avanza el contador p1.

   Orden Total: O(1)+O(1)+O(CP+1)+O(CP*O(CP+1)+O(1)+O(CP*O(1))+O(1))+O(1) = O(CP*CP+CP*CP) = O(2CP^2) = **O(CP^2)**

7. **iRestringir**

   Se define “true” en una coordenada de un subarreglo de una coordenada de un arreglo. Definir tarda O(1).

   Orden Total: O(1)

8. **iPuedeVerR**

   Se obtiene el valor booleano de una coordenada de un subarreglo de una coordenada de un arreglo. Esto tarda O(1). Luego, se aplica “not” a ese valor y se devuelve, también en O(1).

   Orden Total: O(1)

9. **iRestringirLista**

   Se crea una variable de iterador de secuencia y se le asigna el iterador de la secuencia países-esQueLoRestringen, todo en O(1). Luego, se entra a un ciclo que itera una cantidad de veces igual a la cantidad de países de la secuencia países QueLoRestringen. En cada iteración, el ciclo llama a la función restringir, cuya complejidad es O(1), y ésta función llama a Avanzar del iterador de secuencia (O(1)) para obtener el país actual. Luego, se avanza el iterador.

   Orden Total: O(1)+O(1)+O(CP*(O(1+1)+O(1)) = **O(CP)**

10. **iNuevo**

    Se crea un arreglo de tamaño cantidadPaíses en O(CP) y se asigna esto a res en O(1). Se crea una variable nat y se le asigna 0 en O(1). Se entra a un ciclo que itera un número de veces igual a la cantidad de países. En cada iteración, se define en una coordenada del arreglo una tupla de 2 coordenadas (O(2) = O(1)) con dos nuevas secuencias vacías (O(1)) en cada una. Luego, se avanza el contador en O(1).

    Orden Total: O(CP+1)+O(1)+O(1)+O(CP*(O(1+1+1)+O(1))) = **O(CP)**

11. **iRellenar**

    Se crean tres variables nat en O(1) y se asigna 0 a una de ellas, y la longitud del arreglo relaciones (obtenida en O(1)) a otra en O(1). Se entra a un ciclo que itera un número de veces igual a la cantidad de países. En el ciclo, se asigna 0 a la variable nat restante en O(1). Se entra a otro ciclo que también itera un número de veces igual a la cantidad de países. En este ciclo interno, hay un if cuya guarda llama a puedeVer (O(1)), y de ser verdadero, llama a AgregarAtrás de secuencia (O(Copiar un nat) = O(1)) obteniendo estas secuencias del arreglo relaciones en O(1). Luego, se avanza el contador p_2, y fuera de este ciclo, el contador p_1.

    Orden Total: O(1+1+1)+O(1)+O(1)+O(CP*(O(1)+O(CP*(O(1)+O(1+1)+O(1+1))+O(1))+O(1) = O(CP*CP) = **O(CP^2)**

12. **iTieneHueco**

Se crean dos variables nat en O(1). A una se le asigna 0 y a otra la longitud del arreglo de relaciones (obtenida en O(1)), ambas asignaciones costando O(1). Se crean dos variables iteradores de secuencia en O(1), y se asigna "false" a res en O(1). Luego, se entra a un ciclo que itera un número de veces igual a la cantidad de países. Se crean iteradores a las secuencias puedeVer y puedenVerA (en O(1)), obtenidas del arreglo relaciones en O(1). Se entra a un ciclo que itera un número de veces igual a la cantidad de países que hay en la secuencia puedenVer para ese país, y dentro de este ciclo se entra a otro que itera un número de veces igual a la cantidad de países en la secuencia puedenVerA para ese país. En este último ciclo, se entra a un if en cuya guarda se llama a PuedeVer (O(1)) obteniendo los países actuales de cada iterador en O(1), y al valor obtenido se le aplica "not" en O(1). Si es verdadero, se asigna "true" a res en O(1). Luego, se avanza el iterador. Fuera de este ciclo, se avanza el otro iterador, y finalmente se avanza el contador p_1.

Orden Total: O(1+1)+O(1)+O(1+1)+O(1+1)+O(1)+O(CP*(O(1+1)+O(1+1) + O(#PuedeVerpx*(O(#PuedenVerApx*(O(1+1+1+1)+O(1)+O(1))))+O(1))+O(1)) = O (CP*#PuedeVerpx*#PuedenVerApx) = **O($\sum_{(px<longitud(relaciones))}$ #relaciones[px].puedenVerA(px)*#relaciones[px].puedeVer(px))**
"Px" es el país actualmente iterado en la sumatoria.

13. **iTieneAlgunHueco?**

Se crean dos variables nat en O(1) y se le asigna 0 a una de ellas en O(1), y a otra la longitud de las reglas (obtenida en O(1)) en O(1). Se llama a nuevaMatriz (O(CP^2)) y se asigna esta a una variable. Se entra a un ciclo que itera un número de veces igual a la cantidad de países en las reglas. En el ciclo se llama a restringirLista (O(CP)), obteniendo la lista de una coordenada de las reglas en O(1). En el peor caso, son todos los países del sistema. Luego, se avanza el contador. Se crea un nuevo arregloRelaciones (O(CP)) y se llama a rellenar O(CP^2) con ese arregloRelaciones. Finalmente, se asigna a res el resultado de tieneHueco (O($\sum_{(px<longitud(relaciones))}$ #relaciones[px].puedenVerA(px)*#relaciones[px].puedeVer(px))).

Orden Total: O(1)+O(1)+O(1)+O(1+1)+O(CP^2)+O(CP*(O(CP)+O(1))+O(CP)+O(CP^2)+ O($\sum_{(px<longitud(relaciones))}$ #relaciones[px].puedenVerA(px)*#relaciones[px].puedeVer(px)) = O(CP^2)+O(CP*CP)+O(CP)+O(CP^2)+ O($\sum_{(px<longitud(relaciones))}$ #relaciones[px].puedenVerA(px)*#relaciones[px].puedeVer(px)) = O(3CP^2)+O(CP)+ O($\sum_{(px<longitud(relaciones))}$ #relaciones[px].puedenVerA(px)*#relaciones[px].puedeVer(px)) = O(CP^2+O($\sum_{(px<longitud(relaciones))}$ #relaciones[px].puedenVerA(px)*#relaciones[px].puedeVer(px)))
Como la longitud de relaciones es igual a la cantidad de países en las Reglas, y los países que un país puede ver y los que pueden ver a un país son iguales a las funciones PuedeVer y PuedenVerA de las reglas, reemplazando se obtiene:
O(CP^2+$\sum_{(p\in paises(r))}$ #puedenVerA(r,p)*#puedeVer(r,p))

14. **iPuedenVerA**

Se crea un conjunto vacío, lo cual tarda O(1). Se crean también dos variables Nat y se les asigna un valor, también en O(1) (obtener la longitud del vector tarda O(1)). Luego, se entra a un ciclo que itera una cantidad de veces igual a la cantidad de países. En este ciclo, se entra en la guarda de un If en la que se comparan dos nat (O(1)) y se llama a PuedeVer (O(CP)). De ser verdadera esta guarda, se llama a AgregarRápido del conjunto, que tarda el costo de copiar el elemento, pero al ser un Nat, cuesta O(1). Luego se avanza el contador, lo cual cuesta O(1) por ser una operación con Nats. Al salir del ciclo, se devuelve el conjunto.

Orden Total: O(1)+O(1)+O(1+1)+O(CP*(O(1+CP)+O(1)+O(1))) = O(CP*CP) = **O(CP^2)**

15. **iPuedeVer**

Se crea un conjunto vacío, lo cual tarda O(1). Se crean también una variable Nat y se les asigna un valor, también en O(1). Se obtiene una lista de una coordenada, lo cual cuesta O(1), y se crea un iterador a ella, también en O(1). Luego, se entra a un ciclo. Esta ciclo itera hasta recorrer todos los países de la lista. En las guardas de los If del ciclo se hacen comparaciones de Nat, en las que obtener el siguiente del iterador cuesta O(1). Si la primer guarda es verdadera, se llama a AgregarRápido de un nat, lo que cuesta O(1), y se avanza el contador, también en O(1). Si la segunda guarda es verdadera, se avanza el contador en O(1). Si ambas guardas son falsas, se avanza el iterador, también en O(1). El peor caso ocurre cuando la lista tiena a todos los países del sistema. En este caso, se entra a la segunda y tercer guarda una vez por país (ej: el iterador inicia en 0, el contador inicia en 0, la segunda guarda hace avanzar el contador. El iterador sigue en 0, el contador está en 1, la segunda guarda avanza el iterador, etc.).

Orden Total: O(1)+O(1)+O(1+1)+O(CP*(O(1+1)+O(1+1+1)+O(1))+CP*(O(1+1)+O(1+1+1)+O
= O(CP+CP) = O(2CP) = **O(CP)**

3. Módulo diccTrie(clave,significado)

3.1. Interfaz

parámetros formales

géneros diccTrie(α)
usa: bool, puntero, arreglo(α), conj(α)
se explica con: Diccionario(string,α)

Operaciones

VACIO()$\rightarrow$ *res*: **diccTrie(c,s)**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} vacio()\}$
Complejidad: O(1)
Aliasing: No tiene

DEFINIR(**in** *c*: `string`, **in** *s*: `significado`, **in/out** *d*: `diccTrie(s)`)
Pre $\equiv \{d =_{\text{obs}} d_0\}$
Post $\equiv \{d =_{\text{obs}} definir(c, s, d_0) \wedge alias(significado(d, c), s)\}$
Complejidad: O(|c|)
Aliasing: Se genera alias con s en el significado de c. Si se modifica s, se modifica el significado de c.

DEF?(**in** *c*: `string`, **in** *d*: `diccTrie(s)`)$\rightarrow$ *res*: **bool**
Pre $\equiv \{true\}$
Post $\equiv \{res =_{\text{obs}} def?(c, d)\}$
Complejidad: O(|c|)
Aliasing: No tiene

OBTENER(**in** *c*: `string`, **in** *d*: `diccTrie(s)`)$\rightarrow$ *res*: **significado**
Pre $\equiv \{def?(c, d)\}$
Post $\equiv \{res =_{\text{obs}} obtener(c, d) \wedge esAlias(res, significado(d, c)\}$
Complejidad: O(|c|)
Aliasing: res es modificable.

fin interfaz

3.2. Representación

DiccTrie(α) **se representa con** estrDT, donde estrDT es Puntero(Nodo)

Nodo es tupla<*arreglo*: `arreglo(Puntero(Nodo))[27]`, *significado*: `Puntero(`α`)`>

La estructura es un puntero a Nodo en la cual cada nodo es una tupla entre un arreglo y un significado para el dicc. Cada posición del arreglo representa una letra y su contenido es un puntero al nodo de la letra siguiente o a Null.

3.2.1. Invariante de Representación

3.2.1.1. El Invariante Informalmente

1. No hay repetidos en arreglo de Nodo salvo por Null. Todas las posiciones del arreglo están definidas.
2. No se puede volver al Nodo actual siguiendo alguno de los punteros hijo del actual o de alguno de los hijos de estos.
3. O bien el Nodo es una hoja, o todos sus punteros hijo no-nulos llevan a hojas siguiendo su recorrido.

3.2.1.2. El Invariante Formalmente

Rep : estrDT $\rightarrow$ boolean

($\forall e$: `estrDT`) Rep(e) $\equiv$ true $\Longleftrightarrow$

1. $(\forall i, j : \texttt{nat}) 0 \leq i \leq 26 \wedge 0 \leq j \leq 26 \Rightarrow$
   $Definido?((*e).Arreglo, i) \wedge Definido?((*e).Arreglo, j) \wedge$
   $(i = j) \vee$
   $(i \neq j \wedge ((*e).Arreglo[i] = null \wedge (*e).Arreglo[j] = null \vee$
   $(*e).Arreglo[i] \neq (*e).Arreglo[j]) \wedge_{\text{L}}$
2. $(\neg\exists\, n : \texttt{nat}) EncAEstrDTEnNMov(e, e, n) \wedge_{\text{L}}$
3. $SonTodosNullOLosHijosLoSon(e)$

3.2.1.3. Funciones auxiliares

EncAEstrDTEnNMov : estrDT $\times$ estrDT $\times$ Nat $\longrightarrow$ Bool
EncAEstrDTEnNMov(buscado,actual,n) $\equiv$ **if** (n = 0) **then**
EstaEnElArregloActual?(buscado,actual,26)
else
RecurrenciaConLosHijos(buscado,actual, n-1,26)
fi

EstaEnElArregloActual? : estrDT $\times$ estrDT $\times$ nat $\longrightarrow$ Bool
EstaEnElArregloActual?(buscado,actual,n) $\equiv$ **if** (n=0) **then**
((*actual).Arreglo[0] = buscado)
else
((*actual).Arreglo[n] = buscado) $\vee$ (EstaEnElArregloActual? (buscado,actual,n-1))
fi

RecurrenciaConLosHijos : estrDT $\times$ estrDT $\times$ nat $\times$ nat $\longrightarrow$ Bool

RecurrenciaConLosHijos(buscado,actual,n,i) ≡ **if** (i = 0) **then**
EncAEstrDTEnNMov(buscado,(*actual).Arreglo[0],n)
else
EncAEstrDTEnNMov(buscado,
(*actual).Arreglo[i],n) ∨
(RecurrenciaConLosHijos(buscado,actual,n,i-
1)
fi

SonTodosNullOLosHijosLoSon : estrDT ⟶ Bool
SonTodosNullOLosHijosLoSon(e) ≡ Los27SonNull(e,26) ∨ BuscarHijosNull (e, 26)

Los27SonNull : estrDT × nat ⟶ Bool
Los27SonNull(e,i) ≡ **if** (i = 0) **then**
((*e).Arreglo[0] = null)
else
((*e).Arreglo[i] = null) ∧ Los27SonNull(e, i-1)
fi

BuscarHijosNull : estrDT × nat ⟶ Bool
BuscarHijosNull(e,i) ≡ **if** (i = 0) **then**
((*e).Arreglo[0] = null) ∨ SonTodosNullOLosHijosLoSon((*e).Arreglo[0])
else
(((*e).Arreglo[i] = null) ∨ SonTodosNullOLosHijosLo-
Son((*e).Arreglo[i])) ∧ BuscarHijosNull(e,i-1)
fi

3.2.2. Función de Abstracción

Abs : e: `estrDT` → diccT(c,α) Rep(e)

($\forall e$: `estrDT`) Abs(e) =$_{\text{obs}}$ d: diccT(c,α) |

1. ($\forall c$: `clave`)$def?(c,d)$ =$_{\text{obs}}$ $estaDefinido?(c,e)$ $\wedge_{\text{L}}$
2. ($\forall c$: `clave`)$def?(c,d) \Rightarrow obtener(c,d)$ =$_{\text{obs}}$ $ObtenerS(c,*(e))$

3.2.2.1. Funciones auxiliares

estaDefinido? : string × estrDT ⟶ bool
estaDefinido?(c,e) ≡ **if** (e==Null) **then** false **else** NodoDef?(c,*(e)) **fi**

NodoDef? : string × Nodo ⟶ bool
NodoDef?(c,n) ≡ **if** (vacia?(c)) **then**
true
else
if (n.arreglo[numero(prim(c))] ≠ Null) **then**
NodoDef?(fin(c),*(n.arreglo[numero(prim(c))]))
else
false
fi
fi

numero : char $\longrightarrow$ nat
numero(char) $\equiv$ char - a

ObtenerS : string $\times$ Nodo $\longrightarrow \alpha$
ObtenerS(c,n) $\equiv$ **if** (vacia?(c)) **then**
*(n.significado)
else
ObtenerS(fin(c),*(n.arreglo[numero(prim(c))]))
fi

3.3. Algoritmos

Algoritmo 42 iVacio

```
1: function IVACIO→ res: estrDT
2:     var n: Puntero(Nodo)                                    //O(1)
3:     n ← Null                                                //O(1)
4:     res ← n                                                 //O(1)
5: end function
```

Complejidad: O(1)

Algoritmo 43 iDefinir

```
1: function IDEFINIR(in c: string, in s: α, in/out e: estrDT)
2:     if (e = Null) then                                      //O(1)
3:         var n: Nodo                                         //O(1)
4:         n ← iNuevoNodo                                      //O(1)
5:         e ← &(n)                                            //O(1)
6:     end if
7:     var n1: Nodo                                            //O(1)
8:     n1 ← *(e)                                               //O(1)
9:     var i: nat                                              //O(1)
10:    i ← 0                                                   //O(1)
11:    while (i <|c|) do                                       //O(|c|)
12:        if (n1.arreglo[iNumero(c[i])] = Null) then          //O(1)
13:            var n2: Nodo                                    //O(1)
14:            n2 ← iNuevoNodo                                 //O(1)
15:            n1.arreglo[iNumero(c[i])] ← &(n2)               //O(1)
16:        end if
17:        n1 ← *(n1.arreglo[iNumero(c[i])])                   //O(1)
18:        i++                                                 //O(1)
19:    end while
20:    n1.significado ← s                                      //O(1)
21: end function
```

Complejidad: O(|c|)

Algoritmo 44 iNuevoNodo

```
1: function INUEVONODO→ res: Nodo
2:     var n: Nodo                                              //O(1)
3:     n.significado ← Null                                     //O(1)
4:     for (i: nat←0; i<27; i++) do                             //O(27*1)
5:         n.arreglo[i] ← Null                                  //O(1)
6:     end for
7:     res ← n                                                  //O(1)
8: end function
```

Complejidad: O(1)

Algoritmo 45 iNumero

```
1: function INUMERO(c1: char)→ res: nat
2:     var c2: char                                             //O(1)
3:     c2 ← a                                                   //O(1)
4:     res ← (c1 - c2)                                          //O(1)
5: end function
```

Complejidad: O(1)

Nota: Se le resta el char "a" para que al tomar la representación ASCII de los char evaluen como "a=0", "b=1", "c=2", etc.

Algoritmo 46 iDef?

```
1: function IDEF?(in c: string, in e: estr)→ res: bool
2:     if (e ≠ Null) then                                       //O(1)
3:         var n: Nodo                                          //O(1)
4:         n ← *(e)                                             //O(1)
5:         var i: nat                                           //O(1)
6:         var i ← 0                                            //O(1)
7:         res ← true                                           //O(1)
8:         while (i <|c|) do                                    //O(|c|)
9:             if (n.arreglo[numero(c[i])] ≠ Null) then         //O(1)
10:                n ← *(n.arreglo[numero(c[i]))]               //O(1)
11:            else
12:                res ← false                                  //O(1)
13:                i ← long(c)                                  //O(1)
14:            end if
15:            i++                                              //O(1)
16:        end while
17:    else
18:        res ← false                                          //O(1)
19:    end if
20: end function
```

Complejidad: O(|c|)

Algoritmo 47 iObtener

```
1:  function IOBTENER(in c: string, in e: estr)→ res: α
2:      var n: Nodo                                              //O(1)
3:      n ← *(e)                                                 //O(1)
4:      var i: nat                                               //O(1)
5:      var i ← 0                                                //O(1)
6:      while (i <|c|) do                                        //O(|c|)
7:          n ← *(n.arreglo[numero(c[i]))]                       //O(1)
8:          i++                                                  //O(1)
9:      end while
10:     res ← n.significado                                      //O(1)
11: end function
```

Complejidad: O(|c|)

3.4. Analisis de complejidades

1. **iVacio**

   Se crea la variable p de tipo Puntero a Nodo en O(1), luego se le asigna "Null" en O(1) y finalmente se le asigna a res.

   Orden Total: O(1)+O(1)+O(1)=**O(1)**

2. **iDefinir**

   Se evalua si no nada definido y se crea un nuevo Nodo en caso afirmativo, luego se le asigna el puntero a este Nodo a la estrDT. Esto se logra en O(1). Posteriormente se crean algunas variables y se le asignan valores en O(1) y se hace un loop con la longitud del string en O(|string|*O(operaciones dentro del loop)). En el loop se hace un if para evaluar si ya esta definida esa letra y en caso negativo se crea un nuevo nodo y se asigna el puntero a ese nodo. Todo esto se hace en O(1). Luego se asigna al nodo el nodo al cual este apunta en la posición de la letra evaluada y se incrementa el contador del loop. Esto se hace en O(1). Finalmente se asigna al ultimo nodo iterado el significado

   Orden Total: O(1+1+1+1)+O(1+1+1+1)+O(|string|*(O(1+1+1+1+)+O(1+1))+O(1) = O(1)+O(|string|)+O(1) = **O(|string|)**

3. **iNuevoNodo**

   Se crea una variable de tipo Nodo y se le asigna "Null" en O(1). Luego se realiza un For iterando entre 0 y 27 y asignandole a cada posicion del Nodo "Null" en O(1) dando un total para el For de O(27). Finalmente se asigna el nodo a res en O(1).

   Orden Total: O(1+1)+O(27*(O(1)))+O(1)=**O(1)**

4. **iNumero**

   Se crea una variable char y se le asigna un valor en O(1). Luego se asigna a res la resta de 2 chars en O(1). Como res es un nat se asigna el número que representan dichos char.

   Orden Total: O(1)+O(1)+O(1)=**O(1)**

5. **iDef?**

   Se evalua si hay algo definido en O(1). En caso afirmativo se crean variables y se le asignan valor en O(1) y luego se realiza un loop iterando la longitud del string en O(|string|*(operaciones dentro del loop)). Dentro del loop se evalua si esta definido el char correspondiente a la iteración

en y se le asigna al nodo el nodo apuntado en la posición iterada en O(1), caso contrario se asigna "false" a res en O(1). Finalmente incrementa el iterador en O(1).

Orden Total: O(1)+O(1+1+1+1+1)+O(|string|*(O(1+1)+O(1))=**O(|string|)**

6. **iObtener**

   Se crean variables y se les asigna valor en O(1). Luego se realiza un loop iterando la longitud del string en O(|string|*O(operaciones dentro del loop)). Dentro del loop se asigna al nodo el nodo apuntado en la posición iterada y avanza el iterador en O(1). Finalmente se asigna a res el significado en el ultimo nodo asignado en O(1).

   Orden Total: O(1+1+1+1)+O(|string|*O(1+1))+O(1)=**O(|string|)**