UNIVERSIDAD DE BUENOS AIRES

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Algoritmos y Estructuras de Datos II

Trabajo Práctico 2: LinkLinkIt

31 de octubre de 2012

Grupo 10

Integrante	LU	Correo electrónico
Arrejoría, Franco	951/11	francoarrejoria@gmail.com
Maurizio, Miguel Sebastián	635/11	miguelmaurizio.92@gmail.com
Prillo, Sebastián	616/11	sebastianprillo@gmail.com
Tagliavini Ponce, Guido	783/11	guido.tag@gmail.com

Índice

1. Observación

Para poder expresar bien algunas cuestiones de punteros y memoria que están limitadas por el lenguaje de especificación, adoptaremos la siguiente convención acerca de la operación & de punteros. Para que se entienda vamos a dar un ejemplo concreto. Supongamos que tenemos la siguiente estructura:

`estr` es `tupla`(*elPuntero*: `puntero(string)`, *elNombre*: `string`)

y que en el Rep queremos que pedir que *elPuntero* apunte exactamente *elNombre*. Si escribiéramos

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff *e.elPuntero = e.elNombre$

no estaríamos siendo lo suficientemente precisos: puede que elPuntero esté apuntando a otra instancia observacionalmente igual a *elNombre*, pero no a *elNombre* exactamente. Para explicar mejor esta situación, escribiremos

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff e.elPuntero = \&e.elNombre$

En este caso estamos cometiendo el abuso de usar una noción de memoria muy parecida a la del mundo de los bits, pero la encontramos como la mejor forma de ser claros en nuestras afirmaciones del Rep.

2. TAD Iterador de Sistema

TAD Iterador de Sistema

géneros itdesist

exporta itdesist, CrearItDeSist, HayMas?, ProyectarNombre, ProyectarCategoria, ProyectarNumeroAccesos, Avanzar

usa Bool, Nat, Link, Categoria, String, Link de Iterador

igualdad observacional

$(\forall it_1, it_2\text{: itdesist})(it_1 =_{\text{obs}} it_2 \iff (\text{Siguientes}(it_1) = \text{Siguientes}(it_2))$

observadores básicos

Siguientes : itdesist $\longrightarrow$ secu(linkdeit)

generadores

CrearItDeSist : secu(linkdeit)s $\longrightarrow$ itdesist $\{\text{sinLinksRepetidos}(s)\}$

otras operaciones

HayMas?	: itdesist	$\longrightarrow$ bool		
Actual	: itdesistit	$\longrightarrow$ linkdesist		$\{\text{HayMas?}(it)\}$
ProyectarNombre	: itdesistit	$\longrightarrow$ link		$\{\text{HayMas?}(it)\}$
ProyectarCategoria	: itdesistit	$\longrightarrow$ categoria		$\{\text{HayMas?}(it)\}$
ProyectarNumeroAccesos	: itdesistit	$\longrightarrow$ nat		$\{\text{HayMas?}(it)\}$
Avanzar	: itdesistit	$\longrightarrow$ itdesist		$\{\text{HayMas?}(it)\}$
LinksSiguientes	: itdesist	$\longrightarrow$ secu(link)		
SinLinksRepetidos	: secu(linkdeit)	$\longrightarrow$ bool		
NoEsta	: link $\times$ secu(linkdeit)	$\longrightarrow$ bool		
SacarLinks	: secu(linkdeit)	$\longrightarrow$ secu(link)		

axiomas $(\forall it\text{: itdesist})(\forall s\text{: secu(linkdeit)})(\forall l\text{: link})$

Siguientes(CrearItDeSist(s))	$\equiv$	s
HayMas?(it)	$\equiv$	$\neg$ vacia?(Siguientes(it))
Actual(it)	$\equiv$	prim(Siguientes(it))
ProyectarNombre(it)	$\equiv$	Actual(it).nombre
ProyectarCategorias(it)	$\equiv$	Actual(it).categoria
ProyectarNumeroAccesos(it)	$\equiv$	Actual(it).numeroAccesos
Avanzar(it)	$\equiv$	CrearItDeSist(fin(Siguientes(it)))
LinksSiguientes(it)	$\equiv$	SacarLinks(Siguientes(it))

SinLinksRepetidos(s)	$\equiv$	**if** vacia?(s) **then**
		true
		else
		NoEsta(prim(s).nombre, fin(s))$\wedge$ SinLinksRepetidos(fin(s))
		fi

NoEsta(l, s)	$\equiv$	**if** vacia?(s) **then** true **else** ($l \neq$ prim(s).nombre) $\wedge$ NoEsta(l, Fin(s)) **fi**

SacarLinks(s) $\equiv$ **if** vacia?(s) **then** $<>$ **else** prim(s).nombre $\bullet$ SacarLinks(fin(s)) **fi**

Fin TAD

TAD LINK DE ITERADOR **es** TUPLA(nombre: LINK, categoria: CATEGORIA, numeroAccesos: NAT)

géneros linkdeit

3. Módulo DiccionarioTrie(σ)

Interfaz

parámetros formales
géneros σ

se explica con: DICCIONARIO(STRING, σ).

géneros: `diccTrie`(σ).

Operaciones básicas

DEFINIDO?(**in** d: `diccTrie`(σ)), **in** k: `string`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(k, d)}
Complejidad: $O(|k|)$
Descripción: Devuelve true si y sólo si la clave k esta definida en d.

OBTENER(**in** d: `diccTrie(`σ`)`, **in** k: `string`) $\rightarrow$ res : σ
Pre $\equiv$ {def?(k, d)}
Post $\equiv$ {esAlias(res, obtener(k, d))}
Complejidad: $O(|k|)$
Descripción: Devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.

VACIO() $\rightarrow$ res : `diccTrie`(σ)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacío}
Complejidad: $O(1)$
Descripción: Genera un diccionario vacío.

DEFINIR(**in/out** d: `diccTrie`(σ), **in** k: `string` , **in** s: σ)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(k, s, d_0)}
Complejidad: $O(|k| + \text{copy}(s))$
Descripción: Define la clave k con el significado s en d.

Nota: copy es una función de σ en $\mathbb{N}$, que computa el costo temporal de copiar un elemento del género σ.

Representación

`diccTrie`(σ) **se representa con** `estr`

donde `estr` es `puntero(nodo)`
  donde `nodo` es `tupla`(*significado*: `puntero`(σ) , *caracteres*: `arreglo[256] de puntero(nodo)`)

El arreglo caracteres representa cada uno de los caracteres que puede aparecer en una posicion de una clave. La secuencia de caracteres desde la raiz hasta cierto nodo es una clave si y solo si el puntero significado no es `NULL`.

Rep en castellano

1. Todas las posiciones del arrelgo de caracteres están definidas.
2. No hay claves de 0 caracteres. Esto es, el nodo raiz tiene el campo significado `NULL`.
3. No hay ciclos en el trie. Esto es, existe un número natural n tal que la cantidad de niveles del árbol está acotada por n.

4. Dado un nodo cualquiera del trie, existe un único camino desde la raíz hasta dicho nodo.

Notar que si verificamos **3**, entonces para ver **4** basta verificar que en el multiconjunto de punteros no nulos del árbol no hay dos de ellos que apunten al mismo nodo.

Rep formal

Rep : estr $\longrightarrow$ bool

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

(1) ($e \rightarrow \mathit{significado} = \text{NULL}$) $\wedge$

(2) ($\forall i$: nat)($i < 256 \Rightarrow$ definido?(e$\rightarrow$caracteres, i)) $\wedge_{\text{L}}$

(3) ($\exists n$: nat)(finaliza(e, n)) $\wedge_{\text{L}}$

(4) ($\forall p$, q: puntero(nodo))($p \in$ punteros(e) $\wedge$ $q \in$ (punteros(e) $-$ {p}) $\Rightarrow$ $p \neq q$)

finaliza : estr e $\times$ nat $\longrightarrow$ bool $\quad$ {($\forall i$: nat) ($i < 256 \Rightarrow$ definido?($e \rightarrow \mathit{caracteres}, i$))}

finaliza(e, n) $\equiv$ $n > 0 \wedge_{\text{L}}$ ($e =$ NULL $\vee_{\text{L}}$ finalizaAux($e \rightarrow \mathit{caracteres}, n-1, 0$))

finalizaAux : ad(puntero(nodo)) a $\times$ nat $\times$ nat k $\longrightarrow$ bool $\quad$ {$k \leq$ tam(a)}

finalizaAux(a, n, k) $\equiv$ **if** $k =$ tam(a) **then** true **else** finaliza($e \rightarrow \mathit{caracteres}[k], n$) $\wedge$ finalizaAux($a, n, k+1$) **fi**

punteros : estr e $\longrightarrow$ multiconj(puntero(nodo))

punteros(e) $\equiv$ **if** $e =$ NULL **then** $\emptyset$ **else** punterosAux($e \rightarrow \mathit{caracteres}$, 0) **fi**

punterosAux : ad(puntero(nodo)) a $\times$ nat k $\longrightarrow$ multiconj(puntero(nodo)) $\quad$ {$k \leq$ tam(a)}

punterosAux(a, k) $\equiv$ **if** $k =$ tam(a) **then**
$\emptyset$
else
(**if** $a[k] =$ NULL **then** $\emptyset$ **else** Ag($a[k]$, punteros($a[k]$)) **fi**) $\cup$ punterosAux($a, k+1$)
fi

Abs formal

Abs : estr e $\longrightarrow$ diccTrie(σ) $\quad$ {Rep(e)}

Abs(e) $=_{\text{obs}}$ d: diccTrie(σ) | ($\forall c$: string)(def?(c, d) = esClave?(c, e) $\wedge_{\text{L}}$
(def?(c, d) $\Rightarrow_{\text{L}}$obtener(c, d) = significado(c, e)))

esClave? : string c $\times$ estr e $\longrightarrow$ bool $\quad$ {Rep(e)}

esClave?(c, e) $\equiv$ **if** vacía?(c) **then**
$e \rightarrow \mathit{significado} \neq$ NULL
else
$e \rightarrow \mathit{caracteres}$[ord(prim($c$))] $\neq$ NULL $\wedge_{\text{L}}$esClave?(fin(c), $e \rightarrow \mathit{caracteres}$[ord(prim($c$))])
fi

significado : string c $\times$ estr e $\longrightarrow$ σ $\quad$ {Rep(e) $\wedge$ esClave?(c, e)}

significado$(c, e) \equiv$ **if** vacía?(c) **then** $*(e \rightarrow significado)$ **else** significado(fin(c), $e \rightarrow caracteres$[ord(prim(c))]) **fi**

Algoritmos

iDefinido?(**in** d: estr, **in** k: string) $\rightarrow$ res : bool

```
    nat i ← 0       O(1)
    bool esta ← true       O(1)
    puntero(nodo) actual ← d       O(1)
    while i < Longitud(k) ∧ esta  do       O(|k|)
        if actual→ caracteres[ord(k[i])] = NULL  then       O(1)
            esta ← false       O(1)
        end if
        actual ← (actual → caracteres[ord(k[i])])       O(1)
        i← i + 1       O(1)
    end while
    res ← (esta ∧ ¬(actual→ significado = NULL))       O(1)
end function
```

iObtener(**in** d: estr, **in** k: string) $\rightarrow$ res : σ

```
    nat i ← 0       O(1)
    puntero(nodo) actual ← d       O(1)
    while i < Longitud(k) do       O(|k|)
        actual ← (actual→ caracteres[ord(k[i])])       O(1)
        i ← i + 1       O(1)
    end while
    res ← *(actual→ significado)       O(1)
end function
```

iVacio() $\rightarrow$ res : estr

```
    (res→ significado) ← NULL       O(1)
    for i ← 0 to 255 do       O(1)
        (res→ caracteres[i]) ← NULL       O(1)
    end for
end function
```

iDefinir(**in/out** d: estr, **in** k: string, **in** s: σ)

```
    nat i ← 0       O(1)
    puntero(nodo) actual ← d       O(1)
    while i < Longitud(k) do       O(|k|)
        if actual→ caracteres[ord(k[i])] = NULL  then       O(1)
            (actual→ caracteres[ord(k[i])]) ← iVacio()       O(1)
        end if
        actual ← (actual→ caracteres[ord(k[i])])       O(1)
        i ← i + 1       O(1)
    end while
    (actual→ significado) ← &Copiar(s)       O(copy(s))
end function
```

Servicios Usados

El género σ debe proveer una operación:

Copiar(**in** s: σ) $\rightarrow$ res : σ
Pre $\equiv$ {true}

Post $\equiv \{res =_{\text{obs}} s\}$
Complejidad: O(copy(s))

que genera una copia de elemento s, de modo que no haya aliasing entre s y res.

4. Módulo ArbolCategorías

Interfaz

parámetros formales
géneros α

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

géneros: `acat`, `itAcatHijos`, `itAcatPadres`.

Operaciones básicas del árbol

Nuevo(**in** c: `string`) $\rightarrow$ res : `acat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ nuevo(c)}
Complejidad: $O(|c|)$
Descripción: Devuelve un árbol con una única categoría c como raíz.

Raiz(**in** ac: `acat`) $\rightarrow$ res : `string`
Pre $\equiv$ {true}
Post $\equiv$ {esAlias(res, raiz(ac))}
Complejidad: $O(1)$
Descripción: Devuelve la categoría raíz del árbol ac.
Aliasing: La devolución es por referencia. Esta referencia no es modificable.

Agregar(**in/out** ac: `acat`, **in** c: `string`, **in** h: `string`)
Pre $\equiv$ {esta?(c, ac) $\wedge\neg$vacia?(h) $\wedge\neg$esta?(h, ac) $\wedge ac =_{\text{obs}} ac_0$}
Post $\equiv$ {$ac =_{\text{obs}}$ agregar(ac_0, c, h)}
Complejidad: $O(|c| + |h|)$
Descripción: Agrega la categoría c al árbol ac como hijo de h.
Aliasing: h se almacena por copia.

Id(**in** ac: `acat`, **in** c: `string`) $\rightarrow$ res : `nat`
Pre $\equiv$ {esta?(c, ac)}
Post $\equiv$ {$res =_{\text{obs}}$ id(ac, c)}
Complejidad: $O(|c|)$
Descripción: Devuelve el ID de la categoría c en el árbol ac.

CantCats(**in** ac: `acat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ #categorias(ac)}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de categorías que hay en el árbol ac.

Operaciones básicas del iterador de hijos

CrearItHijos(**in** ac: `acat`, **in** c: `string`) $\rightarrow$ res : `itAcatHijos`
Pre $\equiv$ {esta?(c, ac)}
Post $\equiv$ {esPermutacion(Siguientes(res), hijos(ac, c)) $\wedge$ esAlias(Conjunto(Siguientes(res)), hijos(ac, c))}
Complejidad: $O(|c|)$
Descripción: Devuelve un iterador de los hijos directos de la categoría c en el árbol ac.

Nota: la operación Conjunto está axiomatizada abajo.

HayMasHijos(**in** it : `itAcatHijos`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ HayMas?(it)}

Complejidad: $O(1)$
Descripción: Devuelve `true` si y sólo si en el iterador *it* todavía quedan elementos para avanzar.

ActualHijo(**in** *it*: `itAcatHijos`) → *res* : `string`
Pre ≡ {HayMas?(*it*)}
Post ≡ {esAlias(*res*, Actual(*it*))}
Complejidad: $O(1)$
Descripción: Devuelve la categoría en la posición actual del iterador *it*
Aliasing: La devolución es por referencia. Esta referencia no es modificable.

AvanzarHijo(**in/out** *it*: `itAcatHijos`)
Pre ≡ {HayMas?(*it*) $\wedge it =_{\text{obs}} it_0$}
Post ≡ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $O(1)$
Descripción: Avanza el iterador *it* a la posición siguiente.

Operaciones básicas del iterador de padres

CrearItPadres(**in** *ac*: `acat`, **in** *c*: `string`) → *res* : `itAcatPadres`
Pre ≡ {esta?(*c*, *ac*)}
Post ≡ {Siguientes(*res*) = padres(*ac*, *c*)}
Complejidad: $O(|c|)$
Descripción: Crea un iterador unidireccional de todos los padres de una categoría *c* en el árbol *ac*.

Nota: la operación padres está axiomatizada abajo.

HayMasPadres(**in** *it*: `itAcatPadres`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ HayMas?(*it*)}
Complejidad: $O(1)$
Descripción: Devuelve `true` si y sólo si en el iterador *it* todavía quedan elementos para avanzar.

ActualPadre(**in** *it*: `itAcatPadres`) → *res* : `nat`
Pre ≡ {HayMas?(*it*)}
Post ≡ {$res =_{\text{obs}}$ Actual(*it*)}
Complejidad: $O(1)$
Descripción: Devuelve el ID del elemento actual de la posición del iterador *it*.
Aliasing: La devolución es por copia, de modo que no haya aliasing.

AvanzarPadre(**in/out** *it*: `itAcatPadres`)
Pre ≡ {HayMas?(*it*) $\wedge it =_{\text{obs}} it_0$}
Post ≡ {$it =_{\text{obs}}$ Avanzar(it_0)}
Complejidad: $O(1)$
Descripción: Avanza el iterador *it* a la posición siguiente.

Copiar(**in** *it*: `itAcatPadres`) → *res* : `itAcatPadres`
Pre ≡ {true}
Post ≡ {esAlias(Siguientes(*res*), Siguientes(*it*))}
Complejidad: $O(1)$
Descripción: Genera una copia nueva del iterador *it*.

Conjunto : secu(α) $\longrightarrow$ conj(α)

Conjunto(*s*) ≡ **if** vacia?(*s*) **then** $\emptyset$ **else** Ag(prim(*s*), Conjunto(fin(*s*))) **fi**

padres : acat *ac* × categoria *h* $\longrightarrow$ secu(nat) {esta?(*h*, *ac*)}

padres(ac, h) $\equiv$ **if** $h = \text{raiz}(ac)$ **then** vacia **else** id(ac, h) $\bullet$ padres(ac, padre(ac, h)) **fi**

Representación

Representación del árbol

`acat` se representa con `estr`

donde `estr` es `tupla`(*categorias*: `diccTrie(puntero(nodo))` , *cantCats*: `nat` , *raiz*: `string`)

donde `nodo` es `tupla`(*categoria*: `string` , *id*: `nat` , *hijos*: `lista(puntero(nodo))` , *padre*: `puntero(nodo)`)

Cada categoría del árbol está representada por un nodo que contiene su nombre, su ID, un puntero a su nodo padre, y una lista de punteros a sus nodos hijos. El acceso a estos nodos es a través de un diccionario trie de categorías. En este diccionario, el significado de una categoría c es un puntero al nodo del árbol que representa a la categoría c. En otras palabras, de las *hojas* del trie se desprenden los nodos que forman un árbol de categorías.

Rep en castellano

1. Hay al menos una clave en $e.categorias$.
2. $e.cantClaves$ es igual a la cantidad de claves de $e.categorias$.
3. Ningun significado de $e.categorias$ es NULL.
4. Todos los nodos a los que apuntan los significados de $e.categorias$ son distintos.
5. Para cada par n y m de nodos tales que m es hijo de n, si n se obtiene vía un significado de $e.categorias$ entonces m tambien. Más aún, el ID de m es estrictamente mayor que el de n. Notar que esto implica que no hay ciclos en el grafo. Más aún, $m.padre$ es igual a $\&n$.
6. $e.raiz$ esta definido en $e.categorias$ y su significado es un puntero a un nodo con ID igual a 1 y padre NULL.
7. Todos los nodos obtenidos indireccionando los significados significados de $e.categorias$ se pueden obtener vía la raíz (o sea que el grafo inducido por los nodos que nos interesan en conexo).
8. No hay dos nodos en este árbol con un mismo hijo. Esto asegura que en el grafo no hay dos maneras de llegar a un mismo nodo.
9. El significado de una categoría es un puntero a un nodo cuya componente categoría es el nombre de la categoría en cuestión.
10. Los ID de los nodos que se obtienen indireccionando los significados de $e.categorias$ son naturales que toman exactamente los valores entre 1 y $cantCats$ inclusive, o lo que es lo mismo suponiendo el punto **2**, que para cada entero k entre 1 y $cantCats$ inclusive existe una categoría definida en $e.categorias$ cuyo significado sea un puntero a un nodo con ID igual a k.

Rep formal

Rep : estr $\longrightarrow$ bool

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

(1) ($\exists c$: string)(def?($c, e.categorias$))$\wedge$

(2) ($e.cantClaves = \#$(claves($e.categorias$)) $\wedge$

(3) ($\forall c$: string)((def?($c, e.categorias$) $\Rightarrow_{\text{L}}$ obtener($c, e.categorias$) $\neq$ NULL$\wedge$

(4) ($\forall c, d$: string)((def?($c, e.categorias$) $\wedge$ def?($d, e.categorias$) $\wedge$ $c \neq d$) $\Rightarrow_{\text{L}}$ $*$obtener($c, e.categorias$) $\neq$ $*$obtener($d, e.categorias$) $\wedge$

(5) ($\forall c$: string)($\forall n, m$: nodo)(def?($c, e.categorias$) $\wedge_{\text{L}}$ obtener($c, e.categorias$) = $\&n\wedge$ esta?($\&m, n.hijos$)) $\Rightarrow_{\text{L}}$ (($\exists s$: string)(def?($s, e.categorias$) $\wedge_{\text{L}}$ obtener($s, e.categorias$) = $\&m$) $\wedge_{\text{L}}$ $m.id > n.id \wedge m.padre = \&n$)) $\wedge$

(6) (def?($e.raiz$, $e.categorias$) $\wedge_{\text{L}}$ (obtener($e.raiz$, $e.categorias$)$\rightarrow id$ = 1) $\wedge$ (obtener($e.raiz, e.categorias$)$\rightarrow padre$ = NULL)) $\wedge_{\text{L}}$

(7) ($\forall c$: string)(def?($c, e.categorias$) $\wedge\neg(c = e.raiz)$) $\Rightarrow_{\text{L}}$ (($\exists s$: secu(puntero(nodo)))(longitud(s) $> 0 \wedge_{\text{L}}$ obtener($e.raiz, e.categoria$)= prim(s) $\wedge$ obtener($c, e.categorias$)= ult(s) $\wedge$ ($\forall k$: nat)($k <$ longitud(s) - 1 $\Rightarrow_{\text{L}}$ esta?($s[k+1], s[k] \rightarrow hijos$)))) $\wedge$

(8) ($\forall c, d$: string)($\forall n, m, p$: nodo)((def?($c, e.categorias$) $\wedge$ def?($d, e.categorias$) $\wedge_{\text{L}}$ (obtener($c, e.cateogorias$) = $n \wedge$ obtener($d, e.categorias$) = $m\wedge$ esta?($\&p, n.hijos$)) $\wedge$ esta?($\&p, m.hijos$) $\Rightarrow_{\text{L}}$ $m = n$) $\wedge$

(9) ($\forall c$: string) (def?($c, e.categorias$) $\Rightarrow_{\text{L}}$ (obtener($c, e.categorias$)$\rightarrow$ $categoria = c$)) $\wedge$

(10) ($\forall k$: nat) ((1 $\leq$ $k \wedge k \leq e.cantCats$) $\Rightarrow_{\text{L}}$ (($\exists c$: string)(def?($c, e.categorias$) $\wedge_{\text{L}}$ (obtener($c, e.categorias$)$\rightarrow$ $id = k$))))

Abs formal

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

Abs(e) $=_{\text{obs}}$ ac: acat | (categorias(ac = claves($e.categorias$)) $\wedge$
raiz(ac)= $e.raiz \wedge_{\text{L}}$
($\forall h$: categoria)(($h \in$ categorias(ac) $\wedge h \neq$ raiz(ac)) $\Rightarrow_{\text{L}}$
($\exists p$: categoria)($\exists m, n$: nodo)(def?($p, e.categorias$) $\wedge_{\text{L}}$
obtener($p, e.categorias$) = $\&n \wedge$ obtener($h, e.categoria$) = $\&m \wedge$
esta?($\&m, n.hijos$) $\wedge$ padre(ac, h) = $n.categoria$)) $\wedge$
($\forall c$: categoria)($c \in$categorias(ac) $\Rightarrow_{\text{L}}$ (id(ac, c) = (obtener($c, e.categorias$)$\rightarrow$ id)))

Representación del iterador de hijos

`itAcatHijos` **se representa con** `estrItHijos`

donde `estrItHijos` es `itLista(puntero(nodo))`

`itAcatHijos` **se explica con** ITERADOR UNIDIRECCIONAL(STRING)

Rep formal

Rep : estrItHijos $\longrightarrow$ bool

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

Abs formal

Abs : estrItHijos e $\longrightarrow$ itAcatHijos {Rep(e)}

Abs(e) $=_{\text{obs}}$ it: itAcatHijos | Siguientes(it) = secuSiguientes(Siguientes(e))

secuSiguientes : secu(puntero(nodo)) $\longrightarrow$ secu(string)

secuSiguientes(s) $\equiv$ **if** vacia?(s) **then** $<>$ **else** (prim(s) $\rightarrow$ *categoria*) $\bullet$ aString(fin(s)) **fi**

Representación del iterador de padres

`itAcatPadres` **se representa con** `estrItPadres`

donde `estrItPadres` es `puntero(nodo)`

`itAcatPadres` **se explica con** ITERADOR UNIDIRECCIONAL(NAT)

Rep formal

Rep : estrItPadres $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff e \neq$ NULL $\wedge$ ($\exists n$: nat)(finaliza(e, n))

finaliza : estrItPadres e $\times$ nat n $\longrightarrow$ bool

finaliza(e, n) $\equiv$ $n > 0 \wedge_{\text{L}}$ ($e =$ NULL $\vee_{\text{L}}$ finaliza($e \rightarrow padre, n-1$))

Abs formal

Abs : estrItPadres e $\longrightarrow$ itAcatPadres {Rep(e)}

Abs(e) $=_{\text{obs}}$ it: itAcatPadres | Siguientes(it) = secuSiguientes(e)

secuSiguientes : estrItPadres $\longrightarrow$ secu(nat)

secuSiguientes(e) $\equiv$ **if** $e \rightarrow padre =$ NULL **then** $<>$ **else** (($e \rightarrow padre$)$\rightarrow id$)$\bullet$ secuSiguientes($e \rightarrow padre$) **fi**

Algoritmos

Algoritmos del árbol

iNUEVO(**in** c: `string`) $\rightarrow$ *res* : `estr`

```
res.raiz ← COPIAR(c)        O(|c|)
res.cantCats ← 1            O(1)

nodo nuevo                  O(1)
nuevo.id ← 1                O(1)
nuevo.categoria ← COPIAR(c) O(|c|)
nuevo.hijos ← VACIA()       O(1)
nuevo.padre ← NULL          O(1)

res.categorias ← VACIO()    O(1)
```

```
    Definir(res.categorias, c, &nuevo)      O(|c| + copy(&nuevo)) = O(|c|)
end function
```

iRaiz(**in** ac: estr) $\rightarrow$ res : string

```
    res ← ac.raiz      O(1)
end function
```

iAgregar(**in/out** ac: estr, **in** c: string, **in** h: string)

```
    ac.cantCats ← ac.cantCats + 1      O(1)

    puntero(nodo) padre ← Obtener(ac.categorias, c)      O(|c|)

    nodo hijo      O(1)
    hijo.categoria ← Copiar(h)      O(|h|)
    hijo.id ← ac.cantCats      O(1)
    hijo.hijos ← Vacia()      O(1)
    hijo.padre ← padre      O(1)

    AgregarAdelante(alPadre → hijos, &hijo)      O(1)
end function
```

iId(**in** ac: estr, **in** c: string) $\rightarrow$ res : nat

```
    puntero(nodo) alNodo ← Obtener(ac.categorias, c)      O(|c|)
    res ← (alNodo → id)       O(1)
end function
```

iCantCats(**in** ac: estr) $\rightarrow$ res : nat

```
    res ← ac.cantCats      O(1)
end function
```

Algoritmos del iterador de hijos

iCrearItHijos(**in** ac: estr, **in** c: string) $\rightarrow$ res : estrItHijos

```
    puntero(nodo) alNodo ← Obtener(ac.categorias, c)      O(|c|)
    res ← CrearIt(alNodo→ hijos)      O(1)
end function
```

iHayMasHijos(**in** it: estrItHijos) $\rightarrow$ res : bool

```
    res  ← HaySiguiente(it)      O(1)
end function
```

iActualHijo(**in** it: estrItHijos) $\rightarrow$ res : string

```
    res ← (Actual(it)→ categoria)       O(1)
end function
```

iAvanzarHijo(**in/out** it: estrItHijos)

```
    Avanzar(it)      O(1)
end function
```

Algoritmos del iterador de padres

iCrearItPadres(**in** ac: estr, **in** c: string) → res : estrItPadres

```
    res ← Obtener(ac.categorias, c)      O(|c|)
end function
```

iHayMasPadres(**in** it: estrItPadres) → res : bool

```
    res ← ¬((it → padre) = NULL)      O(1)
end function
```

iActualPadre(**in** it: estrItPadres) → res : nat

```
    res ← ((it→ padre)→ id)      O(1)
end function
```

iAvanzarPadre(**in**/**out** it: estrItPadres)

```
    it ← (it→ padre)      O(1)
end function
```

iCopiar(**in** it: estrItPadres) → res : estrItPadres

```
    res ← it      O(1)
end function
```

5. Módulo Sistema

Interfaz

parámetros formales
géneros α

se explica con: LinkLinkIt, Iterador de Sistema.

géneros: `lli`, `itLli`.

Operaciones del sistema

Iniciar(**in** ac: `acat`) $\rightarrow$ res : `lli`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ iniciar(ac) $\wedge$ esAlias(categorias(res), ac)}
Complejidad: $O(\#\text{categorias}(ac))$
Descripción: Devuelve un nuevo sistema. El árbol de categorías ac se guarda por referencia.

Categorias(**in** s: `lli`) $\rightarrow$ res : `puntero(acat)`
Pre $\equiv$ {true}
Post $\equiv$ {esAlias($*res$, categorias(ac))}
Complejidad: $O(1)$
Descripción: Devuelve un puntero al árbol de categorías del sistema s.

NuevoLink(**in/out** s: `lli`, **in** l: `string`, **in** c: `string`)
Pre $\equiv$ {$\neg\ l \in$ links(s) $\wedge$ está?(c, categorias(s)) $\wedge$ $s =_{\text{obs}} s_0$}
Post $\equiv$ {$s =_{\text{obs}}$ nuevoLink(s_0, l, c)}
Complejidad: $O(|l| + |c| + h)$, con $h =$altura(categorias(s)).
Descripción: Agrega el link l al sistema s, en la categoría c.
Aliasing: l y c se almacenan por copia.

Acceso(**in/out** s: `lli`, **in** l: `string`, **in** f: `nat`)
Pre $\equiv$ {$l \in$ links(s) $\wedge$ $f \geq$ fechaActual(s) $\wedge$ $s =_{\text{obs}} s_0$}
Post $\equiv$ {$s =_{\text{obs}}$ acceso(s_0, l, f)}
Complejidad: $O(|l| + h)$, con $h =$altura(categorias(s)).
Descripción: Registra un acceso al link l del sistema s, en la fecha f.

CantLinks(**in** s: `lli`,**in** c: `string`) $\rightarrow$ res : `nat`
Pre $\equiv$ {esta?(c, categorias(s))}
Post $\equiv$ {$res =_{\text{obs}}$ cantLinks(s, c)}
Complejidad: $O(|c|)$
Descripción: Devuelve la cantidad de links asociados a la categoría c o a sus subcategorías, del sistema s.

Operaciones del iterador

CrearIt(**in/out** s: `lli`, **in** c: `string`) $\rightarrow$ res : `itLli`
Pre $\equiv$ {esta?(c, categorias(s)) $\wedge s = s_0$}
Post $\equiv$ {esAlias(linksSiguientes(res), linksOrdenadosPorAccesos(s, c)) $\wedge_L$ ($\forall l$: linkdeit)(esta?(l, Siguientes(res)) $\Rightarrow_L$ $l.categoria =$ categoriaLink(s, $l.nombre$) $\wedge$ $l.numeroAccesos =$ accesosRecientes(s, c, $l.nombre$)) $\wedge$ $s = s_0$}
Complejidad: $O(|c| + n^2)$, con $n =$long(linksOrdenadosPorAccesos(s, c)). Al realizar dos llamadas sucesivas con los mismos argumentos, la segunda cuesta $O(|c| + n)$.
Descripción: Crea un iterador unidireccional de los links de una categoría c del sistema s, ordenados por accesos.

HayMas(**in** it: `itLli`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$HayMas?(it)}

Complejidad: $O(1)$
Descripción: Devuelve `true` si y sólo si en el iterador *it* todavía quedan elementos para avanzar.

PROYECTARLINK(**in** *it*: `itLli`) → *res* : `link`
Pre ≡ {HayMas?(*it*)}
Post ≡ {esAlias(*res*, ProyectarNombre(*it*))}
Complejidad: $O(1)$
Descripción: Devuelve el nombre del link actual de *it*.
Aliasing: La devolución es por referencia. Esta referencia no es modificable.

PROYECTARCATEGORIA(**in** *it*: `itLli`) → *res* : `categoria`
Pre ≡ {HayMas?(*it*)}
Post ≡ {esAlias(*res*, ProyectarCategoria(*it*))}
Complejidad: $O(1)$
Descripción: Devuelve el nombre de la categoría a la cual pertenece el link actual de *it*.
Aliasing: La devolución es por referencia. Esta referencia no es modificable.

PROYECTARNUMEROACCESOS(**in** *it*: `itLli`) → *res* : `nat`
Pre ≡ {HayMas?(*it*)}
Post ≡ {$res =_{\text{obs}}$ ProyectarNumeroAccesos(*it*)}
Complejidad: $O(1)$
Descripción: Devuelve número de accesos recientes del link actual de *it*.

AVANZAR(**in/out** *it*: `itLli`)
Pre ≡ {hayMas?(*it*) $\wedge$ $it =_{\text{obs}} it_0$}
Post ≡ {$it =_{\text{obs}}$ avanzar(it_0)}
Complejidad: $O(1)$
Descripción: Avanza el iterador *it* a la posición siguiente.

Operaciones no exportadas

ACCESOSRECIENTES(**in** *l*: `infoLink`, **in** *ultimo*: `nat`) → *res* : `nat`
Pre ≡ {$l.ultimoAcceso \leq ultimo$}
Post ≡ {$(ultimo - l.ultimoAcceso = 0 \Rightarrow res = l.acceso0 + l.acceso1 + l.acceso2) \wedge$
$(ultimo - l.ultimoAcceso = 1 \Rightarrow res = l.acceso0 + l.acceso1) \wedge$
$(ultimo - l.ultimoAcceso = 2 \Rightarrow res = l.acceso0) \wedge$
$(ultimo - l.ultimoAcceso > 2 \Rightarrow res = 0)$}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de accesos recientes de *l* suponiendo *ultimo* como fecha más reciente.

Representación

Representacion del sistema

`lli` **se representa con** `estr`

donde `estr` es `tupla`(*día*: `nat` , *categorías*: `puntero(acat)` , *links*: `diccTrie(puntero(infoLink))` , *linksDeCats*: `arreglo_dimensionable de lista(puntero(infoLink))` , *linksOrdenados*: `arreglo_dimensionable de puntero(arreglo_dimensionable de puntero(infoLink))`)

donde `infoLink` es `tupla`(*nombre*: `string` , *categoría*: `string` , *idCat*: `nat` , *itPadres*: `itAcatPadres` , *ultimoAcceso*: `nat` , *accesos0*: `nat` , *accesos1*: `nat` , *accesos2*: `nat`)

La estructura más básica que predomina en el sistema es lo que se llama un infoLink, que contiene la información importante de cada link. Un infoLink contiene el nombre del link, el nombre de la categoría a la que pertenece el link, el ID de dicha categoría, un iterador a las categorías superiores, su fecha de último acceso, y tres naturales que indican la cantidad de accesos por cada uno de los últimos 3 días desde la fecha de último acceso a ese link. El acceso a estos

infoLinks se obtiene por varias estructuras.
Recordemos que el sistema se inicia con un ArbolCategorias. A la hora de creación, el sistema guarda un puntero *categorías* a dicho árbol para poder obtener el ID de todas las categorías pertinentes. A excepción de este árbol, todas las demás estructuras del sistema se manejan con los ID de las categorías.
Las dos estructuras importantes que guardan la información de cada categoría son el arreglo *linkDeCats* y el arreglo *linksOrdenados*. La función del arreglo *linksDeCats* consiste en almacenar en la posición $id - 1$, una lista de punteros a todos los links correspondientes a la categoría de ID *id*, y sus subcategorías.
La función del arreglo linksOrdenados es menos obvia. Este arreglo es el que está a cargo de la creación de los iteradores a los links del sistema. La primera vez que se pide el iterador a los links de cierta categoría *id*, se crea un arreglo nuevo de punteros a infoLinks. Este arreglo es el encargado de ordenar, mediante punteros, a los links de la categoría. Una vez realizado este ordenamiento, el arreglo $linksOrdenados[id-1]$ pasa a tener un puntero a este arreglo. De esta manera, si se pide inmediatamente a continuación otra vez el iterador a los links de esa categoría, el sistema nota que los links ya estaban ordenados y devuelve el iterador más rápidamente. Cada vez que se agrega un link a una categoría o se accede a un link, todos los punteros del arreglo *linksOrdenados* correspondientes a esas categorías son puestos en NULL para indicar que el arreglo previamente ordenado ya no es válido. De esta forma, la creación de un nuevo iterador a cualquiera de estas categorías tendrá la complejidad original.
La forma de ubicar a todos los punteros a arreglos que quedarán invalidados en *linksOrdenados* es a través del iterador *itPadres* que contiene cada infoLink. Este iterador proyecta los ID de todas las categorías padres de la categoría del infoLink.
Por último, el sistema cuenta con un diccionario trie *links* de links para tener acceso, vía el nombre de un link, a un puntero a su infoLink correspondiente.
El campo *día* simplemente registra el día actual del sistema.

Rep en castellano

1. Ningún significado de $e.links$ es NULL.
2. El significado de un link l en $e.links$ cumple que su nombre es precisamente l.
3. Para todo infoLink il obtenido vía $e.links$, si $il.ultimoAcceso = 0$ entonces $il.accesos1 = il.accesos2 = 0$, y si $il.ultimoAcceso = 1$ entonces $il.accesos2 = 0$.
4. $e.dia$ es mayor o igual que el último acceso de cualquier link obtenido vía $e.links$.
5. $e.categorias$ no es NULL.
6. Para todo infoLink il que se acceda vía $e.links$, su categoría está definida en $*(e.categorias)$, y su ID, proyectado por $il.idCat$, coincide con su ID en $*(e.categorias)$.
7. $e.linksDeCats[k]$ está definido si y sólo si $0 \leq k \leq \#\text{categorias}(*(e.categorias)) - 1$.
8. Para cada infoLink il tal que $\&il$ pertenece a $e.linksDeCats[k]$, se debe tener que $il.idCat = k + 1$.
9. $e.linksOrdenados[k]$ está definido si y sólo si $0 \leq k \leq \#\text{categorias}(*(e.categorias)) - 1$.
10. Si para la categoria c, $id(*(e.categorias), c) = k$ y $e.linksOrdenados[k-1]$ no es NULL, entonces el arreglo al que apunta tiene tamaño $\#linksCategoriaOHijos(e, c)$.
11. Si $e.linksOrdenados[k]$ está definido y no es NULL, para cada infoLink il tal que $\&il$ pertenece a $*(e.linksOrdenados[k])$, se tiene que $il.idCat = k + 1$.
12. Para todo infoLink il tal que $\&il$ pertenece a $e.linksDeCats[k]$, $il.nombre$ está definido en $e.links$, y su significado es $\&il$.
13. Para todo infoLink il tal que $\&il$ es un significado de $e.links$, existe un k tal que $\&il$ pertenece a $e.linksDeCats[k]$.
14. Para todo infoLink il, si $\&il$ pertenece a $*(e.linksOrdenados[k])$ para algún k, cumple que $il.nombre$ está definido en $e.links$ y su significado es $\&il$.
15. Para todo infoLink il, si $\&il$ pertenece a $*(e.linksOrdenados[k])$ para algún k, entonces $\&il$ pertenece a $e.linksDeCats[k]$.
16. Para todo infoLink il, si $\&il$ pertenece a $e.linksDeCats[k]$ para algún k, entonces $e.linksOrdenados[k]$ es NULL ó $*(e.linksOrdenados[k])$ contiene a $\&il$.

17. Para cada infoLink il, el campo $il.itPadres$ es un iterador que devuelve los ID de las categorías padres de la categorías a la que fue agregada el link. Es decir, siguientes($it.Padres$) recorre los ID de todas estas categorías, y las recorre exactamente una vez.

Observar que en ningún momento incluimos como condición que *linksOrdenados* contenga punteros a arreglos de links que estén ordenados. Podemos considerar, entonces, que estos arreglos simplemente ayudarán a construir el iterador, pero que su contenido no necesariamente obedece a su nombre. Veremos, en los algoritmos, que en la práctica esto no es así, y que siempre que se tenga un puntero no NULL a un arreglo, este estará ordenado.

Rep formal

Rep : estr $\longrightarrow$ bool

$\text{Rep}(e) \equiv \text{true} \iff$ (1) $(\forall l\text{: link})(\text{def?}(l, e.links) \Rightarrow_{\text{L}} \text{obtener}(l, e.links) \neq \text{NULL}) \wedge_{\text{L}}$

(2) $(\forall l\text{: link})(\text{def?}(l, e.links) \Rightarrow_{\text{L}} \text{obtener}(l, e.links) \rightarrow nombre = l) \wedge$

(3) $(\forall l\text{: link})(\forall il\text{: infoLink})((\text{def?}(l, e.links) \wedge_{\text{L}} \text{obtener}(l, e.links) = \&il) \Rightarrow_{\text{L}} ((il.ultimoAcceso = 0 \wedge il.accesos1 = 0 \wedge il.accesos2 = 0) \vee (il.ultimoAcceso = 1 \wedge il.accesos2 = 0) \vee (il.ultimoAcceso \geq 2)))\wedge$

(4) $(\forall l\text{: link})(\forall il\text{: infoLink})((\text{def?}(l, e.links) \wedge_{\text{L}} \text{obtener}(l, e.links) = \&il) \Rightarrow_{\text{L}} (e.dia \geq il.ultimoAcceso)) \wedge$

(5) $e.categorias \neq \text{NULL} \wedge_{\text{L}}$

(6) $(\forall l\text{: link})(\forall il\text{: infoLink})((\text{def?}(l, e.links) \wedge_{\text{L}} \text{obtener}(l, e.links) = \&il) \Rightarrow_{\text{L}} (\text{esta?}(il.categoria, *(e.categorias)) \wedge_{\text{L}} \text{id}(*(e.categorias), il.categoria) = il.idCat) \wedge$

(7) $(\forall k\text{: nat})(\text{definido?}(e.linksDeCats, k) \Leftrightarrow (0 \leq k \wedge k \leq \#\text{categorias}(*(e.categorias))-1)) \wedge$

(8) $(\forall il\text{: infoLink})(\forall k : \text{nat})(\text{esta?}(\&il, e.linksDeCats[k]) \Rightarrow (il.idCat = k + 1)) \wedge$

(9) $(\forall k\text{: nat})(\text{definido?}(e.linksOrdenados, k) \Leftrightarrow (0 \leq k \wedge k \leq \#\text{categorias}(*(e.categorias))-1)) \wedge$

(10) $(\forall c\text{: categoria})(\forall k\text{: nat})((\text{esta?}(c, *(e.categorias)) \wedge_{L} \text{id}(*(e.categorias), c) = k \wedge_{L} linksOrdenados[k-1] \neq \text{NULL}) \Rightarrow_{L} \text{long}(*(e.linksOrdenados[k-1])) = \text{linksCategoriaOHijos}(e, c))$

(11) $(\forall k\text{: nat})(\forall il\text{: infoLink})((\text{definido?}(e.linksOrdenados, k) \wedge_{\text{L}} e.linksOrdenados[k] \neq \text{NULL} \wedge_{\text{L}} \text{esta?}(\&il, *(e.linksOrdenados[k]))) \Rightarrow_{\text{L}} (il.id = k + 1)) \wedge$

(12) $(\forall il\text{: infoLink})(\forall k\text{: nat})((\text{definido?}(e.linksDeCats, k) \wedge_{\text{L}} \text{esta?}(\&il, e.linksDeCats[k])) \Rightarrow_{\text{L}} (\text{def?}(il.nombre, e.links) \wedge_{\text{L}} \text{obtener}(il.nombre, e.links) = \&il)) \wedge$

(13) $(\forall il\text{: infoLink})(\text{def?}(il.nombre, e.links) \Rightarrow_{\text{L}} (\exists k\text{: nat})(\text{definido?}(e.linksDeCats, k) \wedge_{\text{L}} \text{esta?}(\&il, e.linksDeCats[k]))) \wedge$

(14) $(\forall il\text{: infoLink})(\forall k\text{: nat})((\text{definido?}(e.linksOrdenados, k) \wedge_{\text{L}} e.linksOrdenados[k] \neq \text{NULL} \wedge_{\text{L}} \text{esta?}(\&il, *(e.linksOrdenados[k]))) \Rightarrow_{\text{L}} (\text{def?}(il.nombre, e.links) \wedge_{\text{L}} \text{obtener}(il.nombre, e.links) = \&il)) \wedge$

(15) $(\forall il\text{: infoLink})(\forall k\text{: nat})((\text{definido?}(e.linksOrdenados, k) \wedge_{\text{L}} e.linksOrdenados[k] \neq \text{NULL} \wedge_{\text{L}} \text{esta?}(\&il, *(e.linksOrdenados[k]))) \Rightarrow_{\text{L}} (\text{definido?}(e.linksDeCats, k) \wedge_{\text{L}} \text{esta?}(\&il, e.linksDeCats[k]))) \wedge$

(16) $(\forall il\text{: infoLink})(\forall k\text{: nat})((\text{definido?}(e.linksDeCats, k) \wedge_{\text{L}} \text{esta?}(\&il, e.linksDeCats[k])) \Rightarrow_{\text{L}} (e.linksOrdenados[k] = \text{NULL} \vee_{\text{L}} \text{esta?}(\&il, *(e.linksOrdenados[k])))) \wedge_{\text{L}}$

(17) $(\forall l\text{: link})(\forall il\text{: infoLink})((\text{def?}(l, e.links) \wedge_{\text{L}} \text{obtener}(l, e.links) = \&il) \Rightarrow_{\text{L}} (\forall id\text{: nat})(\text{esta?}(id, \text{Siguientes}(il.itPadres)) \Leftrightarrow (\exists c\text{: categoria})(\text{esta?}(c, *(e.categorias)) \wedge_{\text{L}} \text{esSubCategoria}(*(e.categorias), c, il.categoria) \wedge \text{id}(*(e.categorias, c) = id))) \wedge$

(17) $(\forall l\text{: link})(\forall il\text{: infoLink})((\text{def?}(l, e.links) \wedge_{\text{L}} \text{obtener}(l, e.links) = \&il) \Rightarrow_{\text{L}} (\forall i, j\text{: nat}) ((i < \text{long}(\text{Siguientes}(il.itPadres)) \wedge j < \text{long}(\text{Siguientes}(il.itPadres)) \wedge i \neq j) \Rightarrow_{\text{L}} \text{Siguientes}(il.itPadres)[i] \neq \text{Siguientes}(il.itPadres)[j])))$

Abs formal

Representacion del iterador

`itLli` **se representa con** `estrIt`

donde `estrIt` es `tupla`(*arreglo*: `puntero(arreglo_dimensionable de puntero(infoLink))` , *sistema*: `puntero(estr)` , *categoria*: `string` , *tam*: `nat` , *actual*: `nat` , *ultimoAcceso*: `nat`)

Rep en castellano

1. Vale Rep($*(e.sistema)$)
2. $e.arreglo$ no es NULL
3. $e.tam$ es exactamente la dimensión de $*(e.arreglo)$
4. $e.actual$ es menor o igual que $e.tam$ (puede ser igual para denotar que el iterador no tiene siguiente).
5. $e.categoria$ es una categoría del sistema. En otras palabras, $e.categoria$ pertenece al espacio de claves de $*((e.sistema) \rightarrow categorias)$.
6. $e.arreglo$ es exactamente el arreglo $((e.sistema) \rightarrow linksOrdenados)[id - 1]$, donde id es el Id de $e.categoria$ en $e.sistema$.
7. $e.arreglo$ se corresponde con la secuencia linksOrdenadosPorAccesos(Abs($*(e.sistema)$), $e.categoria$).
8. $e.ultimoAcceso$ es la última fecha de acceso entre todos los links de la categoría $e.categoria$ en el sistema $*(e.sistema)$. En el caso en que no hayan links registrados de $e.categoria$ o subcategorías, $e.ultimoAcceso$ debe ser 0.

Notar que aquí estamos pidiendo, a través del punto **6**, que el arreglo del iterador sea uno de los arreglos de *linksOrdenados* del sistema. Al mismo tiempo, por **7**, este arreglo es el de los links ordenados por accesos para la categoría. Esto significa que el arreglo apuntado desde *linksOrdenados* es exactamente el de los links ordenados por accesos.
Dado que en el Rep del sistema, nunca aseguramos esto último, puede sonar contradictorio en un principio. Sin embargo, veremos más adelante, que en el algoritmo encargado de crear el iterador, estamos asegurando esta condición, derivando de allí la validez del predicado Rep.

Rep formal

Rep : estrIt $\longrightarrow$ bool

Rep(e) $\equiv$ true $\iff$ (1) Rep($*(e.sistema)$) $\wedge$

(2) $e.arreglo \neq$ NULL $\wedge_{\text{L}}$

(3) $e.tam =$ tam($*(e.arreglo)$) $\wedge$

(4) $e.actual \leq e.tam$ $\wedge_{\text{L}}$

(5) $e.categoria \in$ claves($*((e.sistema) \rightarrow categorias)$) $\wedge_{\text{L}}$

(6) $e.arreglo = ((e.sistema) \rightarrow linksOrdenados)$[id($*((e.sistema) \rightarrow categorias)$, $e.categoria$) - 1] $\wedge$

(7) secuLinks($*(e.arreglo)$, 0) = linksOrdenadosPorAccesos(Abs($*(e.sistema)$), $e.categoria$) $\wedge_{\text{L}}$

(8) ($\emptyset$(linksCategoriaOHijos(Abs($*(e.sistema)$), $e.categoria$)) $\Rightarrow$ $e.ultimoAcceso$ = 0) $\wedge$ ($\neg\emptyset$(linksCategoriaOHijos(Abs($*(e.sistema)$), $e.categoria$)) $\Rightarrow_{\text{L}}$ $e.ultimoAcceso$ = maximo(diasRecientesParaCategoria(Abs($*(e.sistema)$), $e.categoria$)))

secuLinks : ad(puntero(infoLink)) a × nat n $\longrightarrow$ secu(link) $\{n \leq \text{tam}(a)\}$

secuLinks(a, n) $\equiv$ **if** $n = \text{tam}(a)$ **then** $<>$ **else** $(*(a[n])).nombre \bullet \text{secuLinks}(a, n+1)$ **fi**

maximo : conj(nat) c $\longrightarrow$ nat $\{\neg\emptyset?(c)\}$

maximo(c) $\equiv$ **if** $\#(c) = 1$ **then** dameUno(c) **else** max(dameUno(c), maximo(sinUno(c))) **fi**

Abs formal

Abs : estrIt e $\longrightarrow$ itLli $\{\text{Rep}(e)\}$

Abs(e) $=_{\text{obs}}$ it: itLli | Siguientes(it) = secuSiguientes(e, $e.actual$)

secuSiguientes : estrIt e × nat n $\longrightarrow$ secu(linkDeIt) $\{\text{Rep}(e) \wedge n \leq e.tam\}$

secuSiguientes(e, n) $\equiv$ **if** $n = e.tam$ **then**
$<>$
else
$<((*(e.arreglo))[n])\rightarrow nombre,((*(e.arreglo))[n])\rightarrow categoria,$
$\text{accesos}(*((*(e.arreglo))[n]), e.ultimoAcceso)> \bullet \text{secuSiguientes}(e, n+1)$
fi

accesos : linkDeIt e × nat f $\longrightarrow$ secu(linkDeIt) $\{\text{Rep}(e) \wedge f \leq e.tam\}$

accesos(li, f) $\equiv$ **if** $n - il.ultimoAcceso = 0$ **then**
$l.acceso0 + l.acceso1 + l.acceso2$
else
if $n - il.ultimoAcceso = 1$ **then**
$l.acceso0 + l.acceso1$
else
if $n - il.ultimoAcceso = 2$ **then** $l.acceso0$ **else** 0 **fi**
fi
fi

Algoritmos

Algoritmos del sistema

iInicIar(**in** ac: acat) → res : estr

```
    res.dia ← 0        O(1)
    res.categorias ← &ac        O(1)
    res.links ← Vacio()        O(1)
    nat n ← CantCats(ac)        O(1)
    res.linksDeCats ← CrearArreglo(n)        O(#categorias(ac))
    res.linksOrdenados ← CrearArreglo(n)        O(#categorias(ac))

    for i ← 0 to n - 1 do        O(#categorias(ac))
        res.linksDeCats[i] ← Vacia()        O(1)
        res.linksOrdenados[i] ← NULL        O(1)
    end for
end function
```

iCategorIas(**in** s: estr) → res : puntero(acat)

```
    res ← s.categorias        O(1)
end function
```

iNuevoLInk(**in/out** s: estr, **in** l: string, **in** c: string)

```
    nat id ← Id(*(s.categorias), c)        O(|c|)
```

```
    itAcatPadres it ← CrearItPadres(*(s.categorias), c)      O(|c|)

    infoLink nuevo      O(1)
    nuevo.nombre ← Copiar(l)      O(|l|)
    nuevo.categoria ← Copiar(c)      O(|c|)
    nuevo.idCat ← id      O(1)
    nuevo.itPadres ← Copiar(it)      O(1)
    nuevo.ultimoAcceso ← s.dia      O(1)
    nuevo.accesos0 ← 0      O(1)
    nuevo.accesos1 ← 0      O(1)
    nuevo.accesos2 ← 0      O(1)

    Definir(s.links, l, &nuevo)      O(|l|)
    AgregarAdelante(s.linksDeCats[id - 1], &nuevo)      O(1)
    s.linksOrdenados[id - 1] ← NULL      O(1)

    while HayMasPadres(it) do      O(h)
        id ← Actual(it)      O(1)
        AgregarAdelante(s.linksDeCats[id - 1], &nuevo)      O(1)
        s.linksOrdenados[id - 1] ← NULL      O(1)
        AvanzarPadres(it)      O(1)
    end while
end function
```

Notar que el último ciclo se ejecuta en $O(h)$ ya que it itera sobre las categorías padre de c, las cuales se encuentran, cada una, en un nivel distinto del árbol de categorías. Luego, el ciclo realiza a lo sumo tantas iteraciones como la altura del árbol de categorías.

iAcceso(**in**/**out** s: lli, **in** l: string, **in** f: nat)

```
    puntero(infoLink) il ← Obtener(s.links, l)      O(|l|)
    s.dia ← f      O(1)
    nat u ← (il→ ultimoAcceso)      O(1)

    if f = u then      O(1)
        (il→ accesos0) ← (il→ accesos0) + 1      O(1)
    end if

    if f = u + 1 then      O(1)
        (il→ accesos2) ← (il→ accesos1)      O(1)
        (il→ accesos1) ← (il→ accesos0)      O(1)
        (il→ accesos0) ← 1      O(1)
    end if

    if f = u + 2 then      O(1)
        (il→ accesos2) ← (il→ accesos0)      O(1)
        (il→ accesos1) ← 0      O(1)
        (il→ accesos0) ← 1      O(1)
    end if

    if f ≥ u + 3 then      O(1)
        (il→ accesos2) ← 0      O(1)
        (il→ accesos1) ← 0      O(1)
        (il→ accesos0) ← 1      O(1)
    end if

    (il→ ultimoAcceso) ← f      O(1)
```

```
    nat id ← (il→ idCat)        O(1)
    s.linksOrdenados[id - 1] ← NULL        O(1)

    itAcatPadres it ← Copiar(il→ itPadres)        O(1)

    while HayMasPadres(it) do        O(h)
        id ← Actual(it)        O(1)
        s.linksOrdenados[id - 1] ← NULL        O(1)
        AvanzarPadres(it)        O(1)
    end while
end function
```

Aquí de vuelta, el último ciclo se ejecuta en $O(h)$ por la misma razón de antes.

iCantLinks(**in** s: lli, **in** c: string) → res : nat

```
    nat id ← Id(*(s.categorias), c)        O(|c|)
    res ← Longitud(s.linksDeCats[id - 1])        O(1)
end function
```

Algoritmos del iterador

iCrearIt(**in/out** s: estr, **in** c: string) → res : nat

```
    nat id ← Id(*(s.categorias), c)        O(|c|)
    nat cantLinks ← Longitud(s.linksDeCats[id - 1])        O(1)
    nat res.tam ← cantLinks        O(1)
    arreglo_dimensionable de puntero(infoLink) ordenados        O(1)

    if s.linksOrdenados[id - 1] = NULL then        O(1)
        ordenados ← CrearArreglo(cantLinks)        O(n)
        itLista(puntero(infoLink)) it ← creatIt(s.linksDeCats[id - 1])        O(1)
        for i ← 0 to cantLinks - 1 do        O(n)
            ordenados[i] ← Actual(it)        O(1)
            Avanzar(it)        O(1)
        end for
        s.linksOrdenados[id - 1] ← &ordenados        O(1)
    else
        ordenados ← *((s.linksOrdenados)[id - 1])        O(1)
    end if

    nat ultimo ← 0        O(1)

    for i ← 0 to cantLinks - 1        O(n)
        if ultimo < (*(ordenados[i])).ultimoAcceso then        O(1)
            ultimo ← (*(ordenados[i])).ultimoAcceso        O(1)
        end if
    end for

    res.ultimoAcceso ← ultimo        O(1)

    bool ordenado ← true        O(1)
    for i ← 0 to cantLinks - 2        O(n)
        if iAccesosRecientes(*(ordenados[i]), ultimo) <
            iAccesosRecientes(*(ordenados[i + 1]), ultimo) then        O(1)
            ordenado ← false        O(1)
        end if
```

```
    end for

    if ordenado = false then      O(1)
        for i ← 0 to cantLinks - 2      O(n)
            nat max ← i      O(1)
            for j ← i + 1 to cantLinks - 1      O(n)
                if iAccesosRecientes(*(ordenados[max]), ultimo) <
                    iAccesosRecientes(*(ordenados[j]), ultimo) then      O(1)
                    max ← j      O(1)
                end if
            end for
            puntero(infoLink) aux ← ordenados[max]      O(1)
            ordenados[max] ← ordenados[i]      O(1)
            ordenados[i] ← aux      O(1)
        end for
    end if

    res.arreglo ← &ordenados      O(1)
    res.actual ← 0      O(1)
end function
```

La complejidad termina siendo $O(|c| + n + n^2) = O(|c| + n^2)$.
Observar que si creamos un sistema, y lo hacemos evolucionar según las operaciones provistas, tendremos que si s.linksOrdenados[id - 1] $\neq$ NULL, entonces ello significa que el arreglo de links ordenados por accesos de categoría de ID id, aún tiene vigencia. Esto hace que el chequeo de que el arreglo de links esté ordenado en estos casos sea una cuestión aparentemente innecesaria. Sin embargo, es lo que permite asegurar la correctitud del algoritmo respecto de la especificación y, consecuentemente, el del invariante de representación del iterador creado.

iAccesosRecientes(**in** *l*: infoLink, **in** *ultimo*: nat) → *res* : nat

```
    nat dif ← ultimo - l.ultimoAcceso      O(1)

    if dif = 0 then      O(1)
        res ← l.acceso0 + l.acceso1 + l.acceso2      O(1)
    else if dif = 1 then      O(1)
        res ← l.acceso0 + l.acceso1      O(1)
    else if dif = 2 then      O(1)
        res ← l.acceso0      O(1)
    else
        res ← 0      O(1)
    end if
end function
```

iHayMas(**in** *it*: estrIt) → *res* : bool

```
    res ← (it.actual < it.tam)      O(1)
end function
```

iAvanzar(**in**/**out** *it*: estrIt)

```
    it.actual → it.actual + 1      O(1)
end function
```

iProyectarLink(**in** *it*: estrIt) → *res* : string

```
    res ← ((*(it.arreglo))[it.actual] → nombre)      O(1)
end function
```

iProyectarCategoria(**in** *it*: estrIt) → *res* : string

```
    res ← ((*(it.arreglo))[it.actual] → categoria)        O(1)
end function
```

iProyectarNumeroAccesos(**in** *it*: estrIt) → *res* : string

```
    res ← iAccesosRecientes(*((*(it.arreglo))[actual]), it.ultimoAcceso)        O(1)
end function
```