1. Módulo DiccTrie(string,α)

Interfaz

se explica con: DICCIONARIO(STRING,α).

géneros: `diccTrie`, `itClavesDiccTrie(String)`.

Operaciones básicas de DiccTrie

VACIO() $\rightarrow$ *res* : `diccTrie(string,`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacio()}
Complejidad: $O(1)$
Descripcion: Genera un diccionario vacío

DEFINIDO?(**in** *s*: `string`, **in** *dicc*: `diccTrie(string,`α`)`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ def?(*s*,*dicc*)}
Complejidad: $O(|s|)$
Descripcion: Devuelve true si y sólo si *s* está definido en el diccionario *dicc*.
Aliasing: No se hace aliasing con respecto a la variable *s*.

DEFINIR(**in** *s*: `string`,**in** *info*: α,**in/out** *dicc*: `diccTrie(string,`α`)`)
Pre $\equiv$ {*dicc* $=_{\text{obs}}$ $dicc_0$}
Post $\equiv$ {*dicc* $=_{\text{obs}}$ definir(*s*,*info*,*dicc*)}
Complejidad: $O(|s|)$
Descripcion: Define la clave *s* con el significado *info* en el diccionario *dicc*.
Aliasing: No se hace aliasing con respecto a las variables *s* e *info*.

CLAVES(**in** *dicc*: `diccTrie(string,`α`)`) $\rightarrow$ *res* : `itClavesDiccTrie`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ claves(*dicc*)}
Complejidad: $O(1)$
Descripcion: Devuelve el conjunto de claves del diccionario.
Aliasing: *res* no es modificable.

OBTENER(**in** *s*: `string`, **in** *dicc*: `diccTrie(string,`α`)`) $\rightarrow$ *res* : α
Pre $\equiv$ {def?(*s*,*dicc*)}
Post $\equiv$ {alias(*res* $=_{\text{obs}}$ obtener(*s*,*dicc*))}
Complejidad: $O(|s|)$
Descripcion: Devuelve el significado de la clave *s* en *dicc*.
Aliasing: *res* se pasa por referencia, y es modificable.

CREARITDICC(**in** *c*: `conj(string)`) $\rightarrow$ *res* : `itClavesDiccTrie(string)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(*res*), *c*))}
Complejidad: $O(1)$
Descripcion: Crea un nuevo iterador sobre el conjunto pasado por parámetro.
Aliasing: El iterador se invalida si se elimina el elemento siguiente del iterador. Además Siguientes(*res*) podría cambiar completamente ante cualquier operación que modifique el conjunto.

HAYSIGUIENTE?(**in** *it*: `itClavesDiccTrie(string)`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ haySiguiente(*it*)}
Complejidad: $O(1)$
Descripcion: Devuelve *true* si en el conjunto quedan elementos para iterar.

SIGUIENTE(**in** *it* : `itClavesDiccTrie(string)`) → *res* : `string`
Pre ≡ {true}
Post ≡ {alias(*res* =$_{\text{obs}}$ Siguiente(*it*))}
Complejidad: $O(1)$
Descripcion: Devuelve el elemento al que apunta el iterador.
Aliasing: *res* no es modificable.

AVANZAR(**in/out** *it* : `itClavesDiccTrie(string)`)
Pre ≡ {true}
Post ≡ {*res* =$_{\text{obs}}$ Avanzar(*it*)}
Complejidad: $O(1)$
Descripcion: Avanza a la posición siguiente del iterador.

Representacion

Representación de DiccTrie

`diccTrie` **se representa con** `estrT`

donde `estrT` es `tupla`(*claves*: `conj(string)` , *d*: `diccT`)
donde `diccT` es `tupla`(*hijos*: `arreglo(puntero(diccT))`, *dato*: `puntero(`α`)`)

`itClavesDiccTrie` **se representa con** `icdt`

donde icdt es Tupla(iter: itConj(string))

1. El tamaño de *d.hijos* es 256 (codigo ASCII).
2. Para toda posición del arreglo *d.arbol.hijos* está definida.
3. Los punteros en *d.arbol.hijos* no pueden referise a un nodo anterior de la estructura.
4. *d.claves* contiene el conjunto de claves de todo el arbol.

Rep : diccT $\longrightarrow$ bool
Rep(d) ≡ true $\iff$ (1) tam(d.arbol.hijos)=256 $\wedge_{\text{L}}$
(2)($\forall$ i:nat) $i \leq 255 \Rightarrow$ definido?(d.arbol.hijos, i) $\wedge$
(3)sinRepetidos(punterosNoNulos(d.arbol.hijos)) $\wedge$
(4) d.claves = ObtenerClaves(d.arbol)

punterosNoNulos : arreglo(puntero(diccT)) $\longrightarrow$ arreglo(puntero(diccT))

punterosNoNulos(a) ≡ pNNAux(a,0)

pNNAux : arreglo(puntero(diccT)) × nat $\longrightarrow$ arreglo(puntero(diccT))

pNNAux(a,n) ≡ **if** n = 256 **then**
<>
else
if a[n]=NULL **then**
pNNAux(a,n + 1)
else
a[n]•pNNAux(a,n + 1) & pNNAux(a[n]→hijos,0)
fi
fi

ObtenerClaves : diccT $\longrightarrow$ conj(string)

ObtenerClaves(d) ≡ UnirConjuntos(ObtenerClavesAux(d, 0))

ObtenerClavesAux : diccT × nat $\longrightarrow$ conj(conj(secu(char)))

ObtenerClavesAux(d, i) $\equiv$ **if** $i \leq 255$ **then**
Ag(TodasLasClaves(*(d.hijos)[i], ord$^{-1}(i)$, 0) , ObtenerClavesAux(d, i + 1))
else
$\emptyset$
fi

TodasLasClaves : diccT × char × nat $\longrightarrow$ conj(secu(char))

TodasLasClaves(d, ch, i) $\equiv$ **if** $\neg(d$.hijos[i] = NULL) $\wedge$ (d.dato = NULL) $\wedge$ $i \leq 255$ **then**
TodasLasClaves(*(d.hijos)[i], ch $\bullet$ ord$^{-1}(i)$, i + 1)
else
if $\neg(d$.hijos[i] = NULL) $\wedge$ $\neg(d$.dato = NULL) **then**
Ag($ch \bullet$ ord$^{-1}(i) \bullet <>$, TodasLasClaves(*(d.hijos)[i], $ch \bullet$ ord$^{-1}(i)$, 0))
else
if (d.hijos[i] = NULL) $\wedge$ $\neg(d$.dato = NULL) **then**
Ag($ch \bullet$ ord$^{-1}(i) \bullet <>$, $\emptyset$)
else
$\emptyset$
fi
fi
fi

UnirConjuntos : conj(conj(secu(char))) $\longrightarrow$ conj(secu(char))

UnirConjuntos(cc) $\equiv$ **if** $\neg(\emptyset?(cc))$ **then** dameUno(cc) $\cup$ UnirConjuntos(sinUno(cc)) **else** $\emptyset$ **fi**

Abs : diccT d $\longrightarrow$ diccTrie {Rep(d)}

Abs(d) =$_{\text{obs}}$ d_0: diccTrie | ($\forall$ s:string) def?(s,d_0) $\Leftrightarrow$ hayCamino(s,d_0) $\wedge_L$ def?(s,d_0) $\Rightarrow_L$
obtener(s,d_0) = dameDato(s,d_0)

hayCamino : string × diccT $\longrightarrow$ bool

hayCamino(s,d) $\equiv$ auxHayCamino(StringToArray(s),0,d)

auxHayCamino : arreglo(char) × nat × diccT $\longrightarrow$ bool

auxHayCamino(s,i,d) $\equiv$ **if** i = tam(s) **then**
$\neg$(d$\rightarrow$dato = NULL)
else
if s[i]=NULL **then** false **else** auxHayCamino(s,i+1,d$\rightarrow$hijos[CharToInt(s[i])]) **fi**
fi

dameDato : string s × diccT d $\longrightarrow$ α {hayCamino(s,d)}

dameDato(s,d) $\equiv$ dameDatoAux(StringToArray(s),0,d)

dameDatoAux : arreglo(char) × nat × diccT $\longrightarrow$ α

dameDatoAux(s,n,d) $\equiv$ **if** n = tam(s) **then** *(d.dato) **else** dameDatoAux(s,n+1,d.hijos[CharToInt(s[n])]) **fi**

Algoritmos

iVacio() → *res* : `puntero(diccT)`

arreglo(puntero(diccT)) *sons* ← CrearArreglo(256)	$O(1)$
int $i \leftarrow 0$	$O(1)$
while $i <$256 **do**	$O(1)$
sons[i] ← NULL	$O(1)$
$i \leftarrow i + 1$	$O(1)$
end while	$O(1)$
res.claves ← Vacio()	$O(1)$
res.arbol.hijos ← sons	$O(1)$
res.dato ← NULL	$O(1)$
Complejidad Total:	$O(1)$

Justificación de la complejidad: Crear un arreglo de 256 posiciones es $O(256)$, pero al ser esta una constante, la complejidad de crear el arreglo es $O(1)$. Lo mismo pasa con el while que itera 256 veces, la complejidad es $256 * O(1)$ que es igual a $O(1)$. Luego por ser suma de$O(1)$, la complejidad total es $O(1)$.

iDefinido?(**in** s : `string`, **in** d : `estrT`) → *res* : `bool`

arreglo(char) *word* ← StringToArray(s)	$O(\|s\|)$
int $i \leftarrow 0$	$O(1)$
puntero(diccTrie) *recorre* ← &d.arbol	$O(1)$
while $\neg$(*recorre* = NULL) $\wedge$ $i <$Tam(s) **do**	$O(\|s\|)$
recorre ← *recorre* → hijos[Ord(word[i])]	$O(1)$
end while	
res ← $\neg$(recorre = NULL)	$O(1)$
Complejidad Total:	$O(\|s\|)$

Justificación de la complejidad: *stringToArray*(s) tiene complejidad $|s|$, y el while recorre el arreglo creado por esta función, por lo cual itera $|s|$ veces. Como en cada iteración, realiza una operación con complejidad $O(1)$, la complejidad del while es $|s| * O(1)$, que es igual a $O(|s|)$. Luego, como el resto de las operaciones tienen complejidad $O(1)$, la complejidad total es $O(|s|)$.

iDefinir(**in** s : `string`, **in** $info$: α, **in**/**out** d : `puntero(diccT)`)

arreglo(char) *word* ← StringToArray(s)	$O(\|s\|)$
int $i \leftarrow 0$	$O(1)$
puntero(diccT) *recorre* ← &d.arbol	$O(1)$
while $i <$Tam(s) **do**	$O(\|s\|)$
if (*recorre* → hijos[Ord(word[i])]) = NULL **then**	$O(1)$
puntero(diccT) *nuevo* ← Vacio()	$O(1)$
(*recorre* → hijos[Ord(word[i])) ← *nuevo*	$O(1)$
end if	
recorre ← (*recorre* → hijos[Ord(word[i])])	$O(1)$
end while	
(*recorre* → *dato*) ← &$info$	$O(1)$
Complejidad Total:	$O(\|s\|)$

Justificación de la complejidad: *stringToArray*(s) tiene complejidad $|s|$, y el while recorre el arreglo creado por esta función, por lo cual itera $|s|$ veces. Como en cada iteración, realiza operaciones con complejidad $O(1)$, la complejidad del while es $O(|s|)$. Luego, como el resto de las operaciones tienen complejidad $O(1)$, la complejidad total es $O(|s|)$.

iClaves(**in** d: `estrT`) $\rightarrow$ res : `icdt`

conjunto(string) $c \leftarrow$ &d.claves	$O(1)$
$res \leftarrow$ CrearItDicc(c)	$O(1)$
	Complejidad Total: $O(1)$

iObtener(**in** s: `string`, **in** d: `puntero(diccT)`) $\rightarrow$ res : α

arreglo(char) $word \leftarrow$ StringToArray(s)	$O(\|s\|)$
int $i \leftarrow 0$	$O(1)$
puntero(diccT) $recorre \leftarrow$ &d.arbol	$O(1)$
while $i <$Tam(s) **do**	$O(\|s\|)$
$recorre \leftarrow (recorre \rightarrow$ hijos[Ord(word[i])])	$O(1)$
end while	
$res \leftarrow$ *($recorre \rightarrow$ dato)	$O(1)$
	Complejidad Total: $O(\|s\|)$

Justificación de la complejidad: $stringToArray(s)$ tiene complejidad $|s|$, y el while recorre el arreglo creado por esta función, por lo cual itera $|s|$ veces. Como en cada iteración, realiza una operación con complejidad $O(1)$, la complejidad del while es $|s| * O(1)$, que es igual a $O(|s|)$. Luego, como el resto de las operaciones tienen complejidad $O(1)$, la complejidad total es $O(|s|)$.

iCrearItDicc(**in** c: `conj(string)`) $\rightarrow$ res : `icdt`

itClavesDiccTrie(string) $it \leftarrow$ CrearIt(c)	$O(1)$
res.iter $\leftarrow$ it	$O(1)$
	Complejidad Total: $O(1)$

iHaySiguiente?(**in** it: `icdt`) $\rightarrow$ res : `bool`

$res \leftarrow$ HaySiguiente($icdt$.iter)	$O(1)$
	Complejidad Total: $O(1)$

iSiguiente(**in** it: `icdt`) $\rightarrow$ res : α

$res \leftarrow$ Siguiente($icdt$.iter)	$O(1)$
	Complejidad Total: $O(1)$

iAvanzar(**in/out** it: `icdt`)

Avanzar($icdt$.iter)	$O(1)$
	Complejidad Total: $O(1)$

Servicios Usados

- CrearArreglo(k) con k constante, debe ser $O(1)$.
- StringToArray(s) debe ser $O(|s|)$.
- Ord(c) debe ser $O(1)$.
- Tam(a) debe ser $O(1)$.

2. Módulo ArbolCategorias

Interfaz

se explica con: ArbolCategorias, Iterador Unidireccional, Tupla(nat, nat, string, puntero(nodoCat), conj(puntero(nodoCat))).

géneros: `acat, nodoCat, itCategorias(puntero(nodoCat))`.

Operaciones del Arbol

Nuevo(**in** c: `categoria`) $\rightarrow$ res : `acat`
Pre $\equiv$ {¬vacia?(c)}
Post $\equiv$ {$res =_{\text{obs}}$ nuevo(c)}
Complejidad: $O(|c|)$
Descripcion: Crea un nuevo árbol de categorias.
Aliasing: No se hace aliasing con respecto a c.

Agregar(**in/out** ac: `acat`, **in** c: `categoria`, **in** $hijo$: `categoria`)
Pre $\equiv$ {$ac =_{\text{obs}} ac_0 \wedge$ esta?(c,ac_0) $\wedge$ ¬vacia?($hijo$) $\wedge$ ¬esta?($hijo$,ac_0)}
Post $\equiv$ {$ac =_{\text{obs}}$ agregar(ac_0, c, h)}
Complejidad: $O(|c| + |hijo|)$
Descripcion: Agrega una categoría al árbol de categorias.
Aliasing: No se hace aliasing con respecto a c y a $hijo$.

Categorias(**in** ac: `categoria`) $\rightarrow$ res : `itCategorias`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ categorias(ac)}
Complejidad: $O(1)$
Descripcion: Dado un árbol, devuelve todas sus categorías.
Aliasing: res no es modificable.

Raiz(**in** ac: `acat`) $\rightarrow$ res : `categoria`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ raiz(ac)}
Complejidad: $O(1)$
Descripcion: Devuelve la raíz del arbol.
Aliasing: res no es modificable.

Padre(**in** ac: `acat`, **in** $hijo$: `categoria`) $\rightarrow$ res : `categoria`
Pre $\equiv$ {esta?($hijo$,ac) $\wedge$ ¬($hijo =_{\text{obs}}$ raiz(ac))}
Post $\equiv$ {$res =_{\text{obs}}$ padre(ac,h)}
Complejidad: $O(|hijo| + |padre|)$, donde $padre$ es Padre(ac,$hijo$).
Descripcion: Devuelve el nodo superior de la categoria pasada por parámetro.
Aliasing: res no es modificable y no se hace aliasing con respecto a $hijo$.

Id(**in** ac: `acat`, **in** c: `categoria`) $\rightarrow$ res : `nat`
Pre $\equiv$ {esta?(c,ac)}
Post $\equiv$ {$res =_{\text{obs}}$ id(ac,c)}
Complejidad: $O(|c|)$
Descripcion: Dada una categoría, devuelve su número de id.
Aliasing: no se hace aliasing con respecto a c.

Altura(**in** ac: `acat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ altura(ac)}
Complejidad: $O(1)$
Descripcion: Devuelve la altura del árbol.

EstA?(**in** c: `categoria`, **in** ac: `acat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ esta?(c,ac)}
Complejidad: $O(|c|)$
Descripcion: Devuelve *true* si la categoría pertenece al árbol pasado por parámetro.
Aliasing: no se hace aliasing con respecto a c.

EsSubCategorias(**in** ac: `acat`, **in** pa: `categoria`, **in** hi: `categoria`) $\rightarrow$ res : `bool`
Pre $\equiv$ {esta? (pa, ac) $\wedge$ esta?(hi,ac)}
Post $\equiv$ {$res =_{\text{obs}}$ esSubcategoria(ac,pa,hi)}
Complejidad: $O(|hi| + h)$, donde h es la altura del árbol
Descripcion: Dada una categoría pa y una categoria hi devuelve *true* si hi es subcategoría de pa.
Aliasing: no se hace aliasing con respecto a pa e hi.

AlturaCategoria(**in** ac: `acat`,**in** c: `categoria`) $\rightarrow$ res : `nat`
Pre $\equiv$ {esta(c,ac)}
Post $\equiv$ {$res =_{\text{obs}}$ alturaCategoria(ac,c)}
Complejidad: $O(|c|)$
Descripcion: Devuelve la altura de una categoría.
Aliasing: no se hace aliasing con respecto a c.

Hijos(**in** ac: `acat`,**in** c: `categoria`) $\rightarrow$ res : `itCategorias`
Pre $\equiv$ {esta?(c,ac)}
Post $\equiv$ {$res =_{\text{obs}}$ hijos(ac,c)}
Complejidad: $O(|c|)$
Descripcion: Devuelve las categorías hijas de c.
Aliasing: no se hace aliasing con respecto a c y res no es modificable.

Operaciones del Nodo

IdNodo(**in** n: `nodoCat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $\Pi_1(n)$}
Complejidad: $O(1)$
Descripcion: Dado un nodo, devuelve el id.

AlturaNodo(**in** n: `nodoCat`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $\Pi_2(n)$}
Complejidad: $O(1)$
Descripcion: Dado un nodo, devuelve la altura.

CategoriaNodo(**in** n: `nodoCat`) $\rightarrow$ res : `categoria`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $\Pi_3(n)$}
Complejidad: $O(1)$
Descripcion: Dado un nodo, devuelve el nómbre de la categoría
Aliasing: res no es modificable.

NodoPadre(**in** n: `nodoCat`) $\rightarrow$ res : `nodoCat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $\Pi_4(n)$}
Complejidad: $O(1)$
Descripcion: Dado un nodo, devuelve al nodo padre.
Aliasing: res no es modificable.

TienePadre(**in** n: `nodoCat`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \Pi_1(n) > 1$}
Complejidad: $O(1)$
Descripcion: Devuelve *true* si el nodo pasado por parametro tiene padre

Operaciones del Iterador

CrearItConj(**in** c: `conj(puntero(nodoCat))`) $\rightarrow$ res : `itCategorias`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), c))}
Complejidad: $O(1)$
Descripcion: Crea un nuevo iterador sobre el conjunto pasado por parámetro.
Aliasing: El iterador se invalida si se elimina el elemento siguiente del iterador. Además Siguientes(res) podría cambiar completamente ante cualquier operación que modifique el conjunto.

HaySiguiente?(**in** it: `itCategorias`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ haySiguiente(it)}
Complejidad: $O(1)$
Descripcion: Devuelve *true* si en el iterador quedan elementos para avanzar.

Siguiente(**in** it: `itCategorias`) $\rightarrow$ res : `nodoCat`
Pre $\equiv$ {true}
Post $\equiv$ {alias($res =_{\text{obs}}$ Siguiente(it))}
Complejidad: $O(1)$
Descripcion: Devuelve el elemento al que apunta el iterador.
Aliasing: res no es modificable.

Avanzar(**in/out** it: `itCategorias`)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ Avanzar(it)}
Complejidad: $O(1)$
Descripcion: Avanza a la posición siguiente del iterador.

Representacion

Representación de Arbol de Categorias

`acat` se representa con `estrAc`

donde `estrAc` es `tupla`(*raiz*: `categoria` , *tr*: `diccTrie(categoria,nodoCat)`
, *ultimoId*: `nat` , *alturaArbol*: `nat` , *nodos*: `conjunto(puntero(nodoCat))`)

`nodoCat` se representa con `estrNodo`

donde `nodoCat` es `tupla`(*id*: `nat` , *alturaNodo*: `nat` , *cat*: `categoria` , *padre*: `puntero(nodoCat)`
, *hijos*: `conj(puntero(nodoCat))`)

Invariante de representación en castellano

1. *Raiz* está en las claves de tr.
2. Para todo n en *nodos*, $\&n$ esta en el conjunto de significados de tr.
3. El tamaño de *nodos* es igual al tamaño de el conjunto de significados de tr.
4. Dada una categoría, en su significado, cat es ella misma.
5. El padre de la raiz apunta a NULL.
6. Dada una categoría, en el significado de tr, *cat* de su padre debe estar en las claves de tr.

7. Dada una categoría, en el significado de tr, para cada uno de sus hijos (si el conjunto no es vacio), al buscar su significado en tr, *cat* debe estar en claves de tr.

8. Dada una categoría A, en el significado de tr, *padre*, debe tener a A entre sus hijos, la altura de A es la altura del padre + 1 y el id del padre es estrictamente menor que el de A.

9. Dada una categoría A, en el significado de tr, para cada elemento de *hijos*, A aparece como padre, su altura es la altura de A + 1 y el id es estrictamente mayor que el de A.

10. Existe alguna categoría tal que, en el significado de tr, *alturaNodo* es igual a *alturaArbol*, y para toda otra categoría, en el significado de tr, *alturaNodo* es menor o igual a *alturaArbol*.

11. Existe una categoría tal que, en el significado de tr, *id* es igual a *ultimoId*, y para toda otra categoría, en el significado de tr, *id* es menor estricto que *ultimoId*.

Rep : estrAc $\longrightarrow$ bool
Rep(a) $\equiv$ true $\Longleftrightarrow$ (1) a.raiz $\in$ claves(a.tr) $\wedge$

(2) ($\forall$ n: puntero(nodoCat)) $n \in a$.nodos *Rightarrow* ($\&n \in$ significados(claves(a.tr), a.tr)) $\wedge$

(3) #(a.nodos) = #(significados(claves(a.tr), a.tr) $\wedge$

(4) ($\forall$ c: categoria) def?(c, a.tr) $\Rightarrow$ obtener(c,a.tr).cat = c $\wedge$

(5) obtener(a.raiz, a.tr).padre = NULL $\wedge_{\text{L}}$

(6) ($\forall$ c: categoria) def?(c, a.tr) $\Rightarrow$ (obtener(c,a.tr).padre $\rightarrow$ cat) $\in$ claves(a.tr) $\wedge_{\text{L}}$

(7) ($\forall$ c: categoria) def?(c, a.tr) $\Rightarrow$ (obtener(c,a.tr).hijos).cat $\subseteq$ claves(a.tr) $\wedge_{\text{L}}$

(8) ($\forall$ c: categoria) def?(c, a.tr) $\Rightarrow$ ((c $\in$ (obtener(c,a.tr).padre $\rightarrow$ hijos)) $\wedge$
((obtener(c,a.tr).alturaNodo) = (obtener(c,a.tr).padre $\rightarrow$ alturaNodo) + 1) $\wedge$
((obtener(c,a.tr).id) >(obtener(c,a.tr).padre $\rightarrow$ id)))

(9) ($\forall$ c: categoria) def?(c, a.tr) $\Rightarrow$ ($\forall$ c_h: categoria) ($c_h \in$ obtener(c,a.tr).hijos) $\Rightarrow$
((obtener(c_h,a.tr).padre $\rightarrow$ cat) = c) $\wedge$ ((obtener(c,a.tr).alturaNodo) =
(obtener(c_h,a.tr).alturaNodo) + 1) $\wedge$ ((obtener(c,a.tr).id) <(obtener(c_h,a.tr).id)))

(10) ($\exists$ c: categoria) def?(c, a.tr) $\Rightarrow$ ((obtener(c,a.tr).alturaNodo = a.alturaArbol) $\wedge$
(($\forall$ c_1: categoria)(def?(c_1,a.tr) $\wedge$ c $\neq$ c_1) $\Rightarrow$ (obtener(c_1,a.tr).alturaNodo) $\leq$ a.alturaArbol)) $\wedge$

(11) ($\exists$ c: categoria) def?(c,a.tr) $\Rightarrow$ ((obtener(c,a.tr).id = a.ultimoId) $\wedge$ ($\forall$ c_1: categoria)(def?(c_1,a.tr)
$\wedge$ c $\neq$ c_1) $\Rightarrow$ (obtener(c_1,a.tr).id <a.ultimoId))

significados : conj(string) c $\times$ dicc(string $\times$ nodoCat) d $\longrightarrow$ bool $\{c = claves(d)\}$
significados(c,d) $\equiv$ **if** $c = \emptyset$ **then** $\emptyset$ **else** ag(significado(dameUno(c),d), significados(sinUno(c),d)) **fi**

Abs : estrAc e $\longrightarrow$ acat $\{\text{Rep}(e)\}$
Abs(e) $=_{\text{obs}}$ a: acat | raiz(a) = e.raiz $\wedge$
categorias(a) = claves(e.tr) $\wedge$
($\forall$ h: categoria) (h $\in$ categorias(a)) $\Rightarrow$
((padre(a,h) = (obtener(h,e.tr).padre).cat) $\wedge$ (id(a,h) = obtener(h,e.tr).id))

`itCategorias` se representa con `iuc`

donde itCat es Tupla(c: itConj(puntero(nodoCat)))

Algoritmos

iNuevo(**in** c: `string`) → res : `estrAc`

res.raiz ← Copiar(c)	$O(\|c\|)$
res.ultimoId ← 1	$O(1)$
res.alturaArbol ← 1	$O(1)$
res.tr ← Vacio()	$O(1)$
nodoCat q ← CrearNodo(1,1,c,NULL)	$O(\|c\|)$
definir(c,q,res.tr)	$O(\|c\|)$
conjunto(puntero(nodoCat)) cp ← Vacio()	$O(1)$
res.nodos ← Agregar(cp,&q)	$O(1)$
	Complejidad Total: $O(\|c\|)$

Justificación de la complejidad: $Copar(c)$, $CrearNodo(1, 1, c, NULL)$ y $Definir(c, q, res.tr)$ tienen complejidad, longitud del string que se les pasa por parametro (en este caso |c|). Y como el resto de las operaciones son $O(1)$, la complejidad total del algoritmo es $O(|c|)$.

iAgregar(**in/out** ac: `estrAc`, **in** c: `categoria`, **in** $hijo$: `categoria`)

ac.ultimoId ← ac.ultimoId + 1	$O(1)$
NodoCat q ← Obtener(c,ac.tr)	$O(\|c\|)$
nat $altNodo$	$O(1)$
if (c == Raiz(ac)) **then**	$O(\|c\|)$
$altNodo$ ← 2	$O(1)$
else	
$altNodo$ ← q.alturaNodo + 1	$O(1)$
end if	
if ($altNodo$ >ac.alturaArbol) **then**	$O(1)$
ac.alturaArbol ← $altNodo$	$O(1)$
end if	
NodoCat r ← CrearNodo(ac.ultimoId,$altNodo$,$hijo$,&q)	$O(\|hijo\|)$
definir($hijo$,r,ac.tr)	$O(\|hijo\|)$
nodoCat $nodoHijo$ ← Obtener(ac.tr, $hijo$)	$O(\|hijo\|)$
ac.nodos ← Agregar(ac.nodos,&$nodoHijo$)	$O(1)$
AgregarCatHija(q,$nodoHijo$)	$O(1)$
	Complejidad Total: $O(\|c\| + \|hijo\|)$

iCategorias(**in** ac: `estrAc`) → res : `ItCategorias`

res ← CrearItCategorias(ac.nodos)	$O(1)$
	Complejidad Total: $O(1)$

iRaiz(**in/out** ac: `estrAc`) → res : `categoria`

res ← &ac.raiz	$O(1)$
	Complejidad Total: $O(1)$

iPadre(**in/out** ac: `estrAc`, **in** $hijo$: `categoria`) → res : `string`

nodoCat q ← Obtener($hijo$,ac.tr)	$O(\|hijo\|)$
q ← NodoPadre(q)	$O(1)$
res ← CategoriaNodo(q)	$O(1)$
	Complejidad Total: $O(\|hijo\|)$

iId(**in/out** ac: `estrAc`, **in** c: `categoria`) → res : `nat`

nodoCat q ← Obtener(h,ac.tr)	$O(\|c\|)$
res ← IdNodo(q)	$O(1)$
	Complejidad Total: $O(\|c\|)$

iAltura(**in** ac: estrAc) → res : nat
  $res \leftarrow ac$.AlturaArbol — $O(1)$
  Complejidad Total: $O(1)$

iEsta(**in** ac: estrAc, **in** c: categoria) → res : bool
  if ($c ==$ Raiz(ac)) **then** — $O(|c|)$
    $res \leftarrow$ true — $O(1)$
  else
    $res \leftarrow$ Definido?(c,ac.tr) — $O(|c|)$
  end if
  Complejidad Total: $O(|c|)$

iEsSubCategoria(**in** ac: estrAc, **in** pa: categoria, **in** hi: categoria) → res : bool
  nodoCat $q \leftarrow$ Obtener(hi,ac.tr) — $O(|hi|)$
  bool $esSubCat \leftarrow$ false — $O(1)$
  while ¬(CategoriaNodo(q) == Raiz(ac)) **do** — $O(h)$
    if (CategoriaNodo(q) == pa) **then** — $O(1)$
      $esSubCat \leftarrow$ true — $O(1)$
    end if
    $q \leftarrow$ NodoPadre(q) — $O(1)$
  end while
  $res \leftarrow esSubCat$ — $O(1)$
  Complejidad Total: $O(|hi| + h)$

Justificación de la complejidad: el while itera como mucho h veces, ya que en el peor caso, el nodo sobre el que itera es una hoja del arbol, y como en cada llamado, se le asigna el valor del nodo de su padre, lo maximo que puede recorrer es la atura del árbol. Por otro lado, obtener de c en el diccionario es $O(|c|)$, y como el resto de las funciones tienen complejidad $O(1)$, la complejidad total del algoritmo es $O(|hi| + h)$.

iAlturaCategoria(**in** ac: estrAc,**in** c: categoria) → res : nat
  nodoCat $r \leftarrow$ Obtener(c,ac.tr) — $O(|c|)$
  $res \leftarrow$ AlturaNodo(r) — $O(1)$
  Complejidad Total: $O(|c|)$

iHijos(**in** ac: estrAc,**in** c: categoria) → res : ItCategorias
  nodoCat $r \leftarrow$ Obtener(c,ac.tr) — $O(|c|)$
  $itHijos \leftarrow$ CrearItCategorias(HijosNodo(r)) — $O(1)$
  $res \leftarrow itHijos$ — $O(1)$
  Complejidad Total: $O(|c|)$

CrearNodo(**in** id: nat,**in** $altura$: nat,**in** cat: categoria,**in** $padre$: estrNodo) → res : estrNodo
  res.id $\leftarrow id$ — $O(1)$
  res.alturaNodo $\leftarrow altura$ — $O(1)$
  res.cat $\leftarrow$ Copiar(cat) — $O(|cat|)$
  res.padre $\leftarrow$ &$padre$ — $O(1)$
  res.hijos $\leftarrow$ Vacio() — $O(1)$
  Complejidad Total: $O(|cat|)$

iIdNodo(**in** n: estrNodo) → res : nat
  $res \leftarrow n$.id — $O(1)$
  Complejidad Total: $O(1)$

iAlturaNodo(**in** n: estrNodo) → res : nat
  $res \leftarrow n$.altura — $O(1)$
  Complejidad Total: $O(1)$

iCategoriaNodo(**in** n: estrNodo) → res : string
  $res \leftarrow n$.cat — $O(1)$

Complejidad Total: $O(1)$

iNodoPadre(**in** n : `estrNodo`) → res : `estrNodo`
  $res \leftarrow$ *(n.padre) $O(1)$
Complejidad Total: $O(1)$

iTienePadre(**in** n : `estrNodo`) → res : `bool`
  $res \leftarrow$ (n.padre ≠ NULL) $O(1)$
Complejidad Total: $O(1)$

HijosNodo(**in** n : `estrNodo`) → res : `conj(categoria)`
  $res \leftarrow n$.hijos $O(1)$
Complejidad Total: $O(1)$

AgregarCatHija(**in** n : `estrNodo`, **in** c : `categoria`)
  itUniConj(srting) $it \leftarrow$ CrearItConj(n.hijos) $O(1)$
  $it \leftarrow$ AgregarRapido(n.hijos,c) $O(|c|)$
Complejidad Total: $O(|c|)$

iCrearItCategoria(**in** c : `conj(puntero(nodoCat))`) → res : `itCat`
  itConj(puntero(nodoCat)) $it \leftarrow$ CrearIt(c) $O(1)$
  res.c $\leftarrow it$ $O(1)$
Complejidad Total: $O(1)$

iHaySiguiente?(**in** it : `itCat`) → res : `bool`
  $res \leftarrow$ HaySiguiente?($itCat$.c) $O(1)$
Complejidad Total: $O(1)$

iSiguiente(**in** it : `itCat`) → res : `nodoCat`
  $res \leftarrow$ *Siguiente($itCat$.c) $O(1)$
Complejidad Total: $O(1)$

iAvanzar(**in/out** it : `itCat`)
  Avanzar($itCat$.c) $O(1)$
Complejidad Total: $O(1)$

Observaciones

- CrearNodo, AgregarCatHija e HijosNodo no se exportan

Servicios Usados

- AgregarAdelante(xs,x) y CrearIt(lista) debe ser $O(1)$
- CrearArreglo(n) debe ser $O(n)$
- Copiar(s) debe ser $O(|s|)$
- Obtener(c,d) debe ser $O(|c|)$
- Claves(d) debe ser $O(1)$
- Cardinal(c) debe ser $O(1)$

3. Módulo LINKLINKIT

Interfaz

se explica con: LinkLinkIt.

géneros: `lli`, `itLinks(infoLink)`.

Operaciones de LinkLinkIt

Iniciar(**in** ac: `acat`) → res : `lli`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ iniciar(ac)}
Complejidad: $O(\#Categorias(ac))$
Descripcion: Crea un nuevo sistema.
Aliasing: Se hace aliasing con respecto a la variable ac.

NuevoLink(**in/out** s: `lli`, **in** l: `link`, **in** c: `categoria`)
Pre ≡ {$s =_{\text{obs}} s_0 \wedge \neg(l \in \text{links}(s)) \wedge \text{esta?}(c, \text{categorias}(s))$}
Post ≡ {$s =_{\text{obs}}$ nuevoLink(s_0,l,c)}
Complejidad: $O(|l| + |c| + h)$ donde h es la altura del arbol
Descripcion: Agrega un nuevo link al sistema.
Aliasing: No se hace aliasing con respecto a l ni con respecto a c

Acceso(**in/out** s: `lli`, **in** l: `link`, **in** f: `fecha`)
Pre ≡ {$s = s_0$ y $l \in$ links(s) y $f \geq$ fechaActual(s)}
Post ≡ {$s =_{\text{obs}}$ acceso(s_0,l,f)}
Complejidad: $O(|l| + h)$
Descripcion: Registra las fechas de los accesos a el link dado.
Aliasing: No se hace aliasing con respecto a $link$ ni con respecto a f.

Categorias(**in** s: `lli`) → res : `itCategorias`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ categorias(s)}
Complejidad: $O(1)$
Descripcion: Devuelve las categorias del sistema.
Aliasing: res no es modificable.

Links(**in** s: `lli`) → res : `itLinks`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ links(s)}
Complejidad: $O(CantLinks(Raiz(Categorias(s)) * (|c| + |l|))$ donde c es la categoría con el nombre más largo y l es el link con el nombre más largo.
Descripcion: Devuelve los links del sistema.
Aliasing: res no es modificable.

CategoriaLink(**in** s: `lli`, **in** l: `link`) → res : `categoria`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ categoriaLink (s,l)}
Complejidad: $O(|l|)$
Descripcion: Dado un link, devuelve su categoria.
Aliasing: No se hace aliasing con respecto a r, y res no es modificable.

FechaActual(**in** s: `lli`) → res : `fecha`
Pre ≡ {true}
Post ≡ {$res =_{\text{obs}}$ fechaActual(s)}
Complejidad: $O(1)$
Descripcion: Devuelve la fecha actual del sistema.

FechaUltimoAcceso(**in** s : `lli`, **in** l : `link`) $\rightarrow$ res : `fecha`
Pre $\equiv$ {$l \in$ links(s)}
Post $\equiv$ {$res =_{\text{obs}}$ fechaUltimoAcceso (s,l)}
Complejidad: $O(|l|)$
Descripcion: Dado un link del sistema, devuelve la fecha de su último acceso.
Aliasing: No se hace aliasing con respecto a l.

AccesosRecientesDia(**in** s : `lli`, **in** l : `link`, **in** f : `fecha`) $\rightarrow$ res : `nat`
Pre $\equiv$ {$l \in$ links(s) $\wedge$ esReciente?(s,l,f)}
Post $\equiv$ {$res =_{\text{obs}}$ accesosRecientesDia(s,l,f)}
Complejidad: $O(|l| + Longitud(fya))$ donde fya es una lista con la fecha de lo accesos del link l
Descripcion: Dada una fecha que sea reciente para el link, devuelve los accesos recientes del link para esa fecha.
Aliasing: No se hace aliasing con respecto a l ni con respecto a f.

EsReciente?(**in** s : `lli`, **in** l : `link`, **in** f : `fecha`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$l \in$ links(s)}
Post $\equiv$ {$res =_{\text{obs}}$ esReciente? (s,l,f)}
Complejidad: $O(|l|)$
Descripcion: Dado un link, devuelve true si la fecha pasada por parámetro es reciente para ese link.
Aliasing: No se hace aliasing con respecto a l ni con respecto a f.

AccesosRecientes(**in** s : `lli`, **in** c : `categoria`, **in** l : `link`) $\rightarrow$ res : `nat`
Pre $\equiv$ {esta?(c,categorias(s)) $\wedge$ $l \in$ links(s) $\wedge$ esSubCategoria(categorias(s),c,categoriaLink(s,l))}
Post $\equiv$ {$res =_{\text{obs}}$ accesosRecientes (s,c,l)}
Complejidad: $O(|c| + |l|)$
Descripcion: Dado una categoría y un link de esa categoría, devuelve los accesos recientes para ese link
Aliasing: No se hace aliasing con respecto a c ni con respecto a l.

LinksOrdenadosPorAccesos(**in** s : `lli`, **in** c : `categoria`) $\rightarrow$ res : `itLinks`
Pre $\equiv$ {esta?(c,categorias(s))}
Post $\equiv$ {$res =_{\text{obs}}$ linksOrdenadosPorAccesos (s,c)}
Complejidad: $O(|c| + n)$ en llamadas consecutivas, $O(|c| + n^2)$ en otro caso
Descripcion: Dada una categoría devuelve una lista de links ordenados por cantidad de accesos recientes
Aliasing: No se hace aliasing con respecto a c.

CantLinks(**in** s : `lli`, **in** c : `categoria`) $\rightarrow$ res : `nat`
Pre $\equiv$ {esta?(c,categorias(s))}
Post $\equiv$ {$res =_{\text{obs}}$ cantLinks (s,c)}
Complejidad: $O(|c|)$
Descripcion: Devuelve la cantidad de liks de la categoría c
Aliasing: No se hace aliasing con respecto a c.

CrearItLinks(**in** s : `lli`, **in** l : `lista(infoLink)`) $\rightarrow$ res : `itLinks`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermutacion?(SecuSuby(res), l))}
Complejidad: $O(|l|)$
Descripcion: Crea un iterador que se corresponde con la función que devuelve links ordenados por accesos
Aliasing: res no es modificable.

HayMasLinks(**in** it : `itLinks`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ hayMas?(it)}
Complejidad: $O(1)$
Descripcion: Devuelve *true* si hay mas elementos en la lista para iterar

AvanzarItLinks(**in** it : `itLinks`)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ avanzar(it)}
Complejidad: $O(1)$
Descripcion: Avanza el iterador

ActualItLinks(**in** *it*: `itLinks`) → *res* : `infoLink`
Pre ≡ {true}
Post ≡ {alias(*res* =$_{\text{obs}}$ actual?(*it*))}
Complejidad: $O(1)$
Descripcion: Devuelve el elemento al que está apuntando el iterador
Aliasing: *res* no es modificable.

Representacion

Representación de LinkLinkIt

`lli` **se representa con** `estrSis`

donde `estrSis` es `tupla`(*links*: `diccTrie(link,nodoLink)` , *arbolCat*: `puntero(acat)` , *Ids*: `arregloDimensionable(puntero(nodoIds))` , *accesosSis*: `lista(acceso)` , *linksOrdenados*: `listaDeIterador` , *linksDelSistema*: `lista(infoLink)`)

donde `accesos` es `tupla`(*f*: `fecha` , *link*: `puntero(link)`)

donde `nodoIds` es `tupla`(#*links*: `nat` , *linksCatEHijos*: `puntero(lista(puntero(nodoLink)))` , *diaUltimoAcceso*: `fecha`)

donde `nodoLink` es `tupla`(*AccesosLink*: `lista(fechaYAccesos)` , *cat*: `puntero(categoria)` , *l*: `link`)

donde `fechaYAccesos` es `tupla`(*f*: `fecha` , #*accesos*: `nat`)

donde `listaDeIterador` es `tupla`(*c*: `categoria` , *esLlamadaConsecutiva*: `bool` , *ls*: `lista(puntero(infoLink))`)

`itLinks` **se representa con** `itL`

donde itL es Tupla(it: itLista(infoLink), ls: lista(infoLink))

Invariante de representación en castellano

1. *arbolCat* no es NULL.
2. El tamaño de *ids* es igual a la cantidad de categorias de *arbolCat*.
3. #links de *nodoIds* tiene que ser igual a *Longitud*(*linksCatEHijos*).
4. *diaUltimoAcceso* de *nodoIds* es igual a la fecha f de acceso cuyo link tenga la categoria con id i, siendo i la posicion *ids* donde esta ese *nodoIds*.
5. para todo link de *linksCatEHijos*, ese link pertenece a *claves* del diccTrie links.
6. la categoria de *listaDeIterador* pertenece a las categorias de *arbolCat*.
7. si la lista *ls* de *listaDeIterador* no es vacia, implica que esta ordenada de mayor a menor por *accesosRecientes*.
8. todos los links de esa lista pertenecen al diccTrie *links*, y las categorias a las categorias de *arbolCat*.
9. Para todo link de claves del diccTrie links, la componente l de su significado coincide con la clave,
   9.1. cada posicion de *accesosLink* de *nodoLink* tiene fechas distintas entre si, estan ordenadas de mayor a menor y ademas pertenecen a la lista *accesosSis*,
   9.2. #*accesos* es la suma de los accesos que tuvo el link l en la lista *accesosSis*.
10. si *accesosSis* no es vacia, para cada posicion de la lista, el link pertenece a las claves del diccTrie, y la fecha f es mayor o igual que la fecha siguiente en la lista.
11. Para todo *in* de tipo infoLink, el link está definido en el diccTrie, la categoría es a la que pertenece el link y #Accesos se corresponde con los accesos de ese link.

Rep : estrSis $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$ (1) e.arbolCat $\neq$ NULL $\wedge_{\text{L}}$

(2) tam(e.ids) = #categorias(*(e.arbolCat)) + 1 $\wedge$

(3) ($\forall$ i: nat) $1 \leq i \leq$ tam(e.ids) $\Rightarrow$ ((((e.ids[i]) $\rightarrow$ #links = longitud(e.ids[i] $\rightarrow$ linksCatEHijos)) $\wedge$

(4) ((e.ids[i]) $\rightarrow$ diaUltimoAcceso = diaUltAccParaCatConId(i, e, e.accesosSis))) $\wedge$

(5) ($\forall$ l: link) ($l \in$ (e.ids[i]) $\rightarrow$ linksCatEHijos) $\Rightarrow_{\text{L}}$ ($l \in$ claves(e.links)) $\wedge$

(6) e.linksOrdenados.c $\in$ categorias(*(e.arbolCat)) $\wedge$

(7) ($\neg$vacia?(e.linksOrdenados.ls) $\Rightarrow_{\text{L}}$ (estaOrdenada(e.linksOrdenados.ls)) $\wedge$

(8) losLinksEstanEnElArbol(e.linksOrdenados.ls, e.links)) $\wedge$

(9) ($\forall$ li: link)($li \in$ claves(e.links) $\Rightarrow_{\text{L}}$ li = (obtener(li,e.links)).l $\wedge$

(9.1) fechasDistintasOrdenadasYEnSis(obtener(li, e.links).accesosLink, e.accesosSis) $\wedge$

(9.2) mismaCantAccesos(li, obtener(li, e.links).accesosLink, e.accesosSis)) $\wedge$

(10) ($\neg$vacia?(e.accesosSis) $\Rightarrow_{\text{L}}$ fechasOrdenadasYLinksEnTrie(e.accesosSis, e.links))

(11) ($\forall$ in: infoLink) ($in \in$ e.linksDelSistema) $\Rightarrow_{\text{L}}$ (def?(obtenerLink(in), e. links) $\wedge$ (obtenerCategoriaLink(in) $\in$ categorias(e.arbolCat)) $\wedge$ (obtenerAccesosRecientes (in) = accesosRecientes(s, obtenerLink(in), obtenerCategoriaLink(in))))

diaUltAccParaCatConId : nat $\times$ lli $\times$ secu(nat $\times$ string) $\longrightarrow$ nat

diaUltAccParaCatConId(i,e,ls) $\equiv$ **if** vacia?(ls) **then**
0
else
if id(*(e.arbolCat), categoriaLink(e.prim(ls).l)) = i **then**
prim(ls).f
else
diaUltAccParaCatConId(i,e,finls)
fi
fi

estaOrdenada : secu(infoLink) $\longrightarrow$ bool

estaOrdenada(ls) $\equiv$ **if** longitud(ls) $\leq$ 1 **then**
true
else
if obtenerAccesosRecientes(prim(ls)) $\geq$ obtenerAccesosRecientes (prim(fin(ls))) **then**
estaOrdenada(fin(ls))
else
false
fi
fi

losLinksEstanEnElArbol : secu(infoLink) × dicc(string × nodoLink) $\longrightarrow$ bool

losLinksEstanEnElArbol(ls,d) $\equiv$ **if** vacia?(ls) **then**
true
else
if obtenerLink(prim(ls)) $\in$ claves(d) **then**
losLinksEstanEnElArbol(fin(ls),d)
else
false
fi
fi

fechasDistintasOrdenadasYEnSis : secu(tupla(nat × nat)) × secu(tupla(nat × string)) $\longrightarrow$ bool

fechasDistintasOrdenadasYEnSis(la,ls) $\equiv$ **if** (longitud(la) $\leq$ 1) $\wedge$ (estaFecha?(Π_1(prim(la)),ls)) **then**
true
else
if (Π_1(prim(la)) >Π_1(prim(fin(la))) $\wedge$ (estaFecha?(Π_1(prim(la)),ls))
then
fechasDistintasOrdenadasYEnSis(fin(la),ls)
else
false
fi
fi

estaFecha : nat × secu(tupla(nat × string)) $\longrightarrow$ bool

estaFecha(fe,ls) $\equiv$ **if** vacia?(ls) **then**
false
else
if Π_1(prim(ls)) = fe **then** true **else** estaFecha(fe,fin(ls)) **fi**
fi

mismaCantAccesos : string × secu(tupla(nat × nat)) × secu(tupla(nat × string)) $\longrightarrow$ bool

mismaCantAccesos(l_0,la,ls) $\equiv$ **if** vacia?(la) **then**
true
else
if mismaCantAccesosAux(l_0,la,ls,0) **then**
mismaCantAccesos(l_0,fin(la),ls)
else
false
fi
fi

mismaCantAccesosAux : string × secu(tupla(nat × nat)) × secu(tupla(nat × string)) × nat $\longrightarrow$ bool

mismaCantAccesosAux(l_0,fya,ls,$cont$) $\equiv$ **if** vacia?(ls) **then**
$cont$ = Π_2(fya)
else
if Π_1(fya) <Π_2(prim(ls)) **then**
mismaCantAccesosAux(l_0,fya,fin(ls),cont)
else
if Π_1(fya) = Π_2(prim(ls)) **then**
if Π_2(l_0) **then**
mismaCantAccesosAux(l_0,fya,fin(ls),$cont$ +1)
else
mismaCantAccesosAux(l_0,fya,fin(ls),$cont$)
fi
else
$cont$ = Π_2(fya)
fi
fi
fi

fechasOrdenadasYLinksEnTrie : secu(tupla(nat × string)) × dicc(string × nodoLink) $\longrightarrow$ bool

fechasOrdenadasYLinksEnTrie(la,d) $\equiv$ **if** vacia?(la) **then**
true
else
if longitud(la) = 1 **then**
def?(prim(la).l, d)
else
if prim(la).f $\geq$ prim(fin(la)).f **then**
fechasOrdenadasYLinksEnTrie (fin(la),d)
else
false
fi
fi
fi

accesosRecientes : estrSis × link × categoria $\longrightarrow$ nat

accesosRecientes(s,l,cat) $\equiv$ accesosRecientesAux($*s$.[ids(s.arbolCat(cat))].linksCatEHijos, l,
$*s$.[ids(s.arbolCat(cat))].diaUltimoAcceso)

accesosRecientesAux : lista(puntero(nodoLink) × link × nat $\longrightarrow$ nat

accesosRecientes(ls,l,d) $\equiv$ **if** vacia?(ls) **then**
0
else
if (*prim(ls)).l = l **then**
if (prim(*(prim(ls)).accesosLink).f + 2) $\geq$ d **then**
prim(*(prim(ls)).accesosLink).f + accesosRecientesAux(fin(ls), l, d)
else
accesosRecientesAux(fin(ls), l, d)
fi
else
accesosRecientesAux(fin(ls), l, d)
fi
fi

Abs : estrSis e $\longrightarrow$ LinkLinkIt {Rep(e)}

Abs(e) $=_{\text{obs}}$ s: LinkLinkIt | categorias(s) = Categorias(*(e.arbolCat))
links(s) = claves(e.links)
fechaActual(s) = prim(e.accesosSis).f ($\forall$ l:link)($l \in$ links(s)) tuvoAccesosEnSis(e.accesosSis,l)
$\Rightarrow_L$ (fechaUltimoAcceso(s,l) = Primero(obtener(l, e.links).accesosLink).f
($\forall$ l: link)($\forall$ f: fecha)($l \in$ links(s) $\wedge$ esReciente?(s,l,f)) $\Rightarrow_L$ (accesosRecientesDia(s,l,f)
= ObtenerAccesos(obtener(l, $e.links$).accesosLink, f))

tuvoAccesosEnSis : secu(tupla(nat × string)) × string $\longrightarrow$ bool

tuvoAccesosEnSis(ls,l_0) $\equiv$ **if** vacia?(ls) **then**
false
else
if prim(ls).l = l_0 **then** true **else** tuvoAccesosEnSis(fin(ls),l_0) **fi**
fi

```
obtenerAccesos : secu(tupla(nat × nat)) × nat  ⟶  nat
obtenerAccesos(la,fr) ≡ if vacia?(la) then
                            0
                        else
                            if prim(la).f + 2 ≥ fr then
                                prim(la).#accesos + obtenerAccesos(fin(la), fr)
                            else
                                obtenerAccesos(fin(la), fr)
                            fi
                        fi
```

Algoritmos

iINICIAR(**in** ac: `acat`) → res : `estrSis`

res.arbolCat ← &ac	$O(1)$
res.links ← VACIO()	$O(1)$
res.ids ← CREARARREGLO(CARDINAL(CATEGORIAS(ac)) + 1)	$O(\#Categorias(ac))$
res.ids[0] ← NULL	$O(1)$
res.accesosSis ← VACIA()	$O(1)$
res.linksOrdenados.c ← VACIA()	$O(1)$
res.linksOrdenados.esLlamadaConsecutiva ← false	$O(1)$
res.linksOrdenados.ls ← VACIA()	$O(1)$
nat i ← 1	$O(1)$
while (i <TAM(res.ids)) **do**	$O(\#Categorias(ac))$
nodoIds r	$O(1)$
r.#links ← 0	$O(1)$
r.linksCatEHijos ← VACIA()	$O(1)$
r.diaUltimoAcceso ← 0	$O(1)$
res.ids[i] ← &r	$O(1)$
i ← i + 1	$O(1)$
end while	
Complejidad Total: $O(\#$CATEGORIAS(ac))	

Justificación de la complejidad: Dado un *acat* obtenemos las categorías en $O(\#Categorias(ac))$ por lo tanto creamos el arreglo en O(#Categorias(ac)). En el while, como mucho recorremos $\#Categorias$ por lo tanto, la complejidad es $O(\#Categorias(ac))$. Entonces, la complejidad total del algoritmo es $O(2 * \#Categorias(ac)) = O(\#Categorias(ac))$.

iNuevoLink(**in/out** s: `estrSis`, **in** l: `string`, **in** c: `string`)

nodoLink f_1	
f_1.accesosLink ← Vacia()	$O(1)$
f_1.cat ← &Copiar(c)	$O(1)$
f_1.l ← Copiar(l)	$O(\|l\|)$
Definir(l,f_1,s.links)	$O(\|l\|)$
nodoLink nd ← Obtener(l,s.links)	$O(\|l\|)$
infoLink $info$ ← CrearInfoLink(nd.link, nd.cat, 0)	$O(1)$
AgregarAtras(linksDelSistema, $info$)	$O(1)$
if ($c \neq$ Raiz(s.arbolCat) **then**	$O(\|c\|)$
categoria $catPadre$ ← Padre(c, s.arbolCat)	$O(\|c\|)$
itCategorias it ← Hijos(catPadre, s.arbolCat)	$O(\|c\|)$
puntero(nodoCat) $padre$ ← &(nodoPadre(Siguiente(it)))	$O(1)$
AgregarAdelante((s.ids[Id(s.arbolCat, c)]) → linksCatEHijos), &nd)	$O(1)$
(s.ids[Id(s.arbolCat, c)]) → #Links ← Longitud(ids[Id(s.arbolCat, c)] → linksCatEHijos)	$O(1)$
while TienePadre($padre$) **do**	$O(h)$
nat id ← IdNodo($padre$)	$O(1)$
AgregarAdelante((s.ids[id] → linksCatEHijos), &nd)	$O(1)$
(s.ids[id] → #links) ← Longitud(s.ids[id] → linksCatEHijos)	$O(1)$
$padre$ = NodoPadre(&$padre$)	$O(1)$
end while	
end if	
AgregarAdelante((s.ids[1] → linksCatEHijos), &nd)	$O(1)$
(s.ids[1] → #links) ← Longitud(s.ids[1] → linksCatEHijos)	$O(1)$
s.linksOrdenados.esLlamadaConsecutiva ← false	$O(1)$
Complejidad Total:	$O(\|l\| + \|c\| + h)$

Justificación de la complejidad: El while itera en el peor caso h veces, y como las funciones que se realizan dentro del mismo tienen complejidad $O(1)$, la complejidad el while es $O(h)$. Por otro lado, el resto de las funciones (fuera del while) tienen complejidad $O(|c|)$, $O(|l|)$ y $O(1)$, por lo que al sumar todas, la complejidad total del algoritmo es $O(|l|+|c|+h)$.

iAcceso(**in/out** s: `estrSis`, **in** l: `string`, **in** $fecha$: `nat`)

nodoLink in ← Obtener(l,s.links)	$O(\|l\|)$
if (≠(EsVacia(in.accesosLink)) ∧ (Primero(in.accesosLink).f == $fecha$)) **then**	$O(1)$
Primero(in.accesosLink) ← Primero(in.accesosLink) + 1	$O(1)$
else	
fechaYAccesos fa	$O(1)$
fa.#accesos ← 1	$O(1)$
fa.f ← $fecha$	$O(1)$
AgregarAdelante(in.accesosLink, fa)	$O(copy(fa)) = O(1)$
end if	
acceso acc	$O(1)$
acc.f ← $fecha$	$O(1)$
acc.link ← Copiar(&in.l)	$O(\|l\|)$
AgregarAdelante(s.accesosSis, acc)	$O(copy(acc)) = O(1)$
nat i ← Id(*(in.cat), s.arbolCat)	$O(h)$
((s.ids[i]) → diaUltimoAcceso) ← $fecha$	$O(1)$
s.linksOrdenados.esLlamadaConsecutiva ← false	$O(1)$
Complejidad Total:	$O(\|l\| + h)$

Justificación de la complejidad: La complejidad total del algoritmo es $O(|l| + h)$, por ser suma de funciones con complejidad $O(|l|)$, $O(h)$ y $O(1)$.

iCategorias(**in** s: `estrSis`) → res : `itCategorias`

res ← Categorias(*(s.arbolCat))	$O(1)$
Complejidad Total:	$O(1)$

iLinks(**in** s : estrSis) $\rightarrow$ res : itL

$ls \leftarrow s$.linksDelSistema	$O(1)$
for nat $i \leftarrow 0$ **to** Longitud(ls) **do**	$O(Longitud(ls) * (\|ls.link\| + \|ls[i].cat\|))$
link $l \leftarrow$ *(ls[i].link)	$O(\|ls.link\|)$
categoria $cat \leftarrow$ *(ls[i].cat)	$O(\|ls[i].cat\|)$
ls[i].accesosRec $\leftarrow$ AccesosRecientes(cat, l)	$O(\|cat\| + \|l\|)$
end for	
$res \leftarrow$ CrearItLinks(ls)	$O(1)$
	Complejidad Total: $O(Longitud(ls) * (\|ls.link\| + \|ls[i].cat\|))$

iCategoriaLink(**in** s : estrSis, **in** l : string) $\rightarrow$ res : string

nodoLink $f \leftarrow$ Obtener(l,s.links)	$O(\|l\|)$
$res \leftarrow f$.cat	$O(1)$
	Complejidad Total: $O(\|l\|)$

iFechaActual(**in** s : estrSis) $\rightarrow$ res : fecha

acceso $a \leftarrow$ Primero(s.accesosSis)	$O(1)$
$res \leftarrow a$.f	$O(1)$
	Complejidad Total: $O(1)$

iFechaUltimoAcceso(**in** s : estrSis, **in** l : string) $\rightarrow$ res : nat

nodoLink $i \leftarrow$ Obtener(l,s.links)	$O(\|l\|)$
$res \leftarrow$ Primero(i.accesosLink).f	$O(1)$
	Complejidad Total: $O(\|l\|)$

iAccesosRecientesDia(**in** s : estrSis, **in** l : string, **in** $fecha$: nat) $\rightarrow$ res : nat

nodoLink $in \leftarrow$ Obtener(l,s.links)	$O(\|l\|)$
lista(fechaYAccesos) $fya \leftarrow in$.accesosLink	$O(1)$
itLista(fechaYAccesos) $it \leftarrow$ CrearIt(fya)	
while (HaySiguiente(it)) **do**	$O(Longitud(fya))$
if (Siguiente(it).f == $fecha$) **then**	
$res \leftarrow$ Siguiente(it).#accesos	$O(1)$
else	
Avanzar(it)	$O(1)$
end if	
end while	
	Complejidad Total: $O(\|l\| + Longitud(fya))$

Justificación de la complejidad: El while itera sobre la cantidad de elementos que hay en fya, y como las funciones que se encuentran dentro del mismo tienen complejidad $O(1)$, la complejidad del while es $O(Longitud(fya))$. Y como el resto de las funciones del algoritmo tienen complejidad $O(|l|)$ y $O(1)$, la complejidad total del algoritmo es $O(|l| + Longitud(fya))$

iEsReciente(**in** s : estrSis, **in** l : string, **in** $fecha$: nat) $\rightarrow$ res : bool

$res \leftarrow$ (MenorReciente(s,l) $\leq fecha$) $\wedge$ ($fecha \leq$ FechaUltimoAcceso(s,l))	$O(\|l\|)$
	Complejidad Total: $O(\|l\|)$

MenorReciente(**in** s : estrSis, **in** l : string) $\rightarrow$ res : nat

nat $res \leftarrow$ Max(FechaUltimoAcceso(s,l) + 1, diasRecientes) - diasRecientes	$O(\|l\|)$
	Complejidad Total: $O(\|l\|)$

Max(**in** n : nat, **in** m : nat) $\rightarrow$ res : nat

if $n > m$ **then**	$O(1)$
$res \leftarrow n$	$O(1)$
else	
$res \leftarrow m$	$O(1)$
end if	
	Complejidad Total: $O(1)$

```
iAccesosRecientes(in s: estrSis, in c: string, in l: string) → res : nat
  nat recientes ← DiasRecientesParaCategoria(s,c)                                   O(|c|)
  nat accesos ← 0                                                                   O(1)
  nodoLink info ← Obtener(l,s.links)                                                O(|l|)
  itLista(fechaYAccesos) itF ← CrearIt(info.accesosLink)                            O(1)
  for nat i ← 0 to 2 do
    if (HaySiguiente(itF)) then                                                     O(1)
      nat fecha ← Siguiente(itf).f                                                  O(1)
      if (recientes ≤ fecha + 2) then                                               O(1)
        nat accesosDeEseLink ← Siguiente(itF).#accesos                              O(1)
        accesos ← accesos + accesosDeEseLink                                        O(1)
      end if
      Avanzar(itF)                                                                  O(1)
    end if
    res ← accesos                                                                   O(1)
  end for
                                                  Complejidad Total: O(|c| + |l|)

AccesosRecientesRapido(in s: estrSis, in recientes: nat, in nl: nodoLink) → res : nat
  nat accesos ← 0                                                                   O(1)
  itLista(fechaYAccesos) itf ← CrearIt(nl.accesosLink)                              O(1)
  for nat i ← 0 to 2 do                                                             O(1)
    if (HaySiguiente)(itF) then nat fecha ← Siguiente(itF).f                        O(1)
      if (recientes ≤ fecha + 2) then                                               O(1)
        nat accesosDeEseLink ← Siguiente(itF).#accesos                              O(1)
        accesos ← accesos + accesosDeEseLink                                        O(1)
      end ifAvanzar(itF)
    end if
  end for
  res ← accesos                                                                     O(1) x
                                                                                    O(1)

DiasRecientesParaCategoria(in s: estrSis, in c: string) → res : nat
  nat i ← Id(*(s.arbolCat),c)                                                       O(|c|)
  res ← (s.ids[i]) → diaUltimoAcceso                                                O(1)
                                                  Complejidad Total: O(|c|)

iLinksOrdenadosPorAccesos(in s: estrSis, in c: string) → res : itL
  if (s.linksOrdenados.c = cat) ∧ (s.linksOrdenados.esLlamadaConsecutiva) then     O(|c|)
    res ← CrearItLinks(s.linksOrdenados.ls)                                         O(n)
  else
    nat i ← Id(c, *(s.arbolCat))                                                    O(|c|)
    lista(puntero(nodoLink)) listaLinks ← (s.ids[i]) → linksCatEHijos               O(1)
    nat d ← (s.ids[i]) → diaUltimoAcceso                                            O(1)
    OrdenarEnElLugar(s,listaLinks,d)                                                O(n^2)
    s.linksOrdenados.c ← copiar(cat)                                                O(|c|)
    s.linksOrdenados.esLlamadaConsecutiva ← true                                    O(1)
    PasarAListaInfoLink(s,listaLinks,d,linksOrdenados.ls)                           O(n)
    res ← CrearItLinks(s.linksOrdenados.ls)                                         O(1)
  end if
                                  Complejidad Total: O(|c| + n) en llamadas consecutivas
                                  Complejidad Total: O(|c| + n^2) en otro caso
```

OrdenarEnElLugar(**in/out** s: `estrSis`, **in** ls: `lista(puntero(nodoLink))`, **in** $diaUltimoAcceso$: `nat`)

itLista(puntero(nodoLink)) $it \leftarrow$ CrearIt(ls)	$O(1)$
for (nat $i \leftarrow 0$ **to** Longitud(ls) - 1) **do**	$O(\sum_{i=0}^{Longitud(ls)} O(Longitud(ls))) = O(Longitud(ls)^2)$
itLista(puntero(nodoLink)) $max \leftarrow it$	$O(1)$
itLista(puntero(nodoLink)) $it_0 \leftarrow it$	$O(1)$
Avanzar(it_0)	$O(1)$
for (nat $j = i + 1$ **to**Longitud(ls)) **do**	$O(\sum_{i=0}^{Longitud(ls)} O(1)) = O(Longitud(ls))$
if (**then**AccesosRecientesRapido(s,$diaUltimoAcceso$,*(Siguiente(it_0)) >	
AccesosRecientesRapido(s,$diaUltimoAcceso$,*(Siguiente(max)))	$O(1)$
$max \leftarrow it_0$	$O(1)$
end if	
Avanzar(it_0)	
end for	
puntero(nodoLink) $aux \leftarrow$ Siguiente(it)	$O(1)$
Siguiente(it) $\leftarrow$ Siguiente(max)	$O(1)$
Siguiente(max) $\leftarrow aux$	$O(1)$
Avanzar(ti)	$O(1)$
end for	
	Complejidad Total: $O(Longitud(ls)^2)$

iCantLinks(**in** s: `estrSis`, **in** c: `categoria`) $\rightarrow res$: `nat`

nat $i \leftarrow$ Id(*(s.arbolCat),c)	$O(\|c\|)$
$res \leftarrow (s.\text{ids}[i]) \rightarrow$ #links	$O(1)$
	Complejidad Total: $O(\|c\|)$

iCrearItLinks(**in/out** s: `estrSis`, **in** l: `lista(infoLink)`) $\rightarrow res$: `itL`

res.ls $\leftarrow$ Copiar(l)	$O(\|l\|)$
res.it $\leftarrow$ CrearIt(res.ls)	$O(1)$
	Complejidad Total: $O(\|l\|)$

iAvanzarItLinks(**in/out** it: `itL`)

Avanzar(it)	$O(1)$
	Complejidad Total: $O(1)$

iActualItLinks(**in** it: `itL`) $\rightarrow res$: `infoLink`

$res \leftarrow$ Actual(it)	$O(1)$
	Complejidad Total: $O(1)$

iHayMasItLinks(**in** it: `itL`) $\rightarrow res$: `bool`

$res \leftarrow$ HayMas?(it)	$O(1)$
	Complejidad Total: $O(1)$

PasarAListaInfoLink(**in** s: `estrSis`, **in** $recientes$: `nat`,
in $listaLinks$: `lista(puntero(nodoLink))`, **in** ls: `lista(infoLink)`)

lista(puntero(infoLink)) $lin \leftarrow$ Vacia()	$O(1)$
itLista(puntero(nodoLink)) $it \leftarrow$ CrearIt($listaLinks$)	$O(1)$
while HaySiguiente(it) **do**	$O(Longitud(listaLinks))$
nat $accesosRec \leftarrow$ AccesosRecientesRapido($recientes$,*Siguiente(it))	$O(1)$
infoLink $in \leftarrow$ tupla(Siguiente(it) $\rightarrow$ link, Siguiente(it) $\rightarrow$ cat, accesosRec)	$O(1)$
AgregarAtras(lin,in)	$O(1)$
Avanzar(it)	$O(1)$
end while	
$ls \leftarrow lin$	$O(1)$
	Complejidad Total: $O(Longitud(listaLinks))$

Justificación de la complejidad: El while itera sobre la cantidad de elementos que hay en $listaLinks$, y como todas las funciones que estan dentro del mismo tienen complejidad $O(1)$, la complejidad del while es $O(Longitud(listaLinks))$. Como el resto de las funciones tienen complejidad $O(1)$, la complejidad total es $O(Longitud(listaLinks))$.

Servicios Usados

- VACIA() debe ser $O(1)$
- CREARARREGLO(n) debe ser $O(n)$
- CARDINAL(c) debe ser $O(1)$
- TAM(ls) debe ser $O(1)$
- CREARIT(ls) debe ser $O(1)$
- SIGUIENTE(it) debe ser $O(1)$
- AGREGARATRAS(ls,e) debe ser $O(1)$

4. Módulo InfoLink

Interfaz

se explica con: TUPLA(STRING, STRING, NAT).

géneros: `infoLink`.

Operaciones básicas de infoLink

CREARINFOLINK(**in** l: `link`, **in** c: `categoria`, **in** acc: `nat`) $\rightarrow$ res : `infoLink`
Pre $\equiv$ {$\neg$vacio(l) $\wedge$ $\neg$vacio(c)}
Post $\equiv$ {$res =_{\text{obs}} \langle l, c, acc \rangle$}
Complejidad: $O(1)$
Descripcion: Crea un nuevo $infoLink$.
Aliasing: Hace aliasing con respecto a l y con respecto a c.

OBTENERLINK(**in** $info$: `infoLink`) $\rightarrow$ res : `link`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \Pi_1(info)$}
Complejidad: $O(1)$
Descripcion: Dado un $infoLink$ devuelve un link.
Aliasing: res no es modificable.

OBTENERCATEGORIALINK(**in** $info$: `infoLink`) $\rightarrow$ res : `categoria`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \Pi_2(info)$}
Complejidad: $O(1)$
Descripcion: Dado un $infoLink$ devuelve una categoría.
Aliasing: res no es modificable.

OBTENERACCESOSRECIENTES(**in** $info$: `infoLink`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} \Pi_3(info)$}
Complejidad: $O(1)$
Descripcion: Devuelve el conjunto de claves del diccionario.

Representacion

Representación de InfoLink

`infoLink` **se representa con** `estrInfo`

donde `estrInfo` es `tupla`(l: `puntero(link)` , c: `puntero(categoria)` , $accesosRec$: `nat`)

Rep : estrInfo $\longrightarrow$ bool
Rep(e) $\equiv$ true $\iff$ $\neg$vacio((*e.l)) $\wedge$ $\neg$vacio(*(e.c))

Abs : estrInfo e $\longrightarrow$ infoLink {Rep(e)}
Abs(e) $=_{\text{obs}}$ i: infoLink | $\Pi_1(i) = e$.l $\wedge$ $\Pi_2(i) = e$.c $\wedge$ $\Pi_3(i) = e$.accesosRec

Algoritmos

iCrearInfoLink(**in** l : `link`, **in** c: `categoria`, **in** acc: `nat`) $\rightarrow$ res : `estrInfo`

res.l $\leftarrow$ &l	$O(1)$
res.c $\leftarrow$ &c	$O(1)$
res.accesosRec $\leftarrow$ acc	$O(1)$
	Complejidad Total: $O(1)$

iObtenerLink(**in** $info$: `estrInfo`) $\rightarrow$ res : `string`

$res \leftarrow$ *($info$.l)	$O(1)$
	Complejidad Total: $O(1)$

iObtenerCategoriaLink(**in** $info$: `estrInfo`) $\rightarrow$ res : `string`

$res \leftarrow$ *($info$.c)	$O(1)$
	Complejidad Total: $O(1)$

iObtenerAccesosRecientes(**in** $info$: `estrInfo`) $\rightarrow$ res : `string`

$res \leftarrow$ $info$.accesoRec	$O(1)$
	Complejidad Total: $O(1)$